Practice Test (Python) - 1

Question 1: **Correct**

Which of the following options describes the responsibility of the executors in Spark?

- ○ **The executors accept jobs from the driver, analyze those jobs, and return results to the driver.**
- ○ **The executors accept tasks from the driver, execute those tasks, and return results to the cluster manager.**
- ◉ **The executors accept tasks from the driver, execute those tasks, and return results to the driver.** (Correct)
- ○ **The executors accept tasks from the cluster manager, execute those tasks, and return results to the driver.**
- ○ **The executors accept jobs from the driver, plan those jobs, and return results to the cluster manager.**

Question 2: **Correct**

Which of the following describes the role of tasks in the Spark execution hierarchy?

- ◉ **Tasks are the smallest element in the execution hierarchy.** (Correct)
- ○ **Within one task, the slots are the unit of work done for each partition of the data.**
- ○ **Tasks are the second-smallest element in the execution hierarchy.**
- ○ **Stages with narrow dependencies can be grouped into one task.**
- ○ **Tasks with wide dependencies can be grouped into one stage.**

Question 3: **Correct**

Which of the following describes the role of the cluster manager?

- ○ **The cluster manager schedules tasks on the cluster in client mode.**
- ○ **The cluster manager schedules tasks on the cluster in local mode.**
- ◉ **The cluster manager allocates resources to Spark applications and maintains the executor processes in client mode.** (Correct)
- ○ **The cluster manager allocates resources to Spark applications and maintains the executor processes in remote mode.**
- ○ **The cluster manager allocates resources to the DataFrame manager.**

Question 4: **Correct**

Which of the following is the idea behind dynamic partition pruning in Spark?

- ◉ **Dynamic partition pruning is intended to skip over the data you do not need in the results of a query.** (Correct)
- ○ **Dynamic partition pruning concatenates columns of similar data types to optimize join performance.**
- ○ **Dynamic partition pruning performs wide transformations on disk instead of in memory.**
- ○ **Dynamic partition pruning reoptimizes physical plans based on data types and broadcast variables.**
- ○ **Dynamic partition pruning reoptimizes query plans based on runtime statistics collected during query execution.**

Question 5: **Correct**

Which of the following is one of the big performance advantages that Spark has over Hadoop?

- ○ **Spark achieves great performance by storing data in the DAG format, whereas Hadoop can only use parquet files.**
- ○ **Spark achieves higher resiliency for queries since, different from Hadoop, it can be deployed on Kubernetes.**
- ◉ **Spark achieves great performance by storing data and performing computation in memory, whereas large jobs in Hadoop require a large amount of relatively slow disk I/O operations.** (Correct)
- ○ **Spark achieves great performance by storing data in the HDFS format, whereas Hadoop can only use parquet files.**
- ○ **Spark achieves performance gains for developers by extending Hadoop's DataFrames with a user-friendly API.**

Question 6: **Incorrect**

Which of the following is the deepest level in Spark's execution hierarchy?

- ○ **Job**
- ○ **Task** (Correct)
- ◉ **Executor** (Incorrect)
- ○ **Slot**
- ○ **Stage**

Question 7: **Incorrect**

Which of the following statements about garbage collection in Spark is incorrect?

- ○ **Garbage collection information can be accessed in the Spark UI's stage detail view.**
- ○ **Optimizing garbage collection performance in Spark may limit caching ability.**
- ○ **Manually persisting RDDs in Spark prevents them from being garbage collected.** (Correct)
- ○ **In Spark, using the G1 garbage collector is an alternative to using the default Parallel garbage collector.**
- ◉ **Serialized caching is a strategy to increase the performance of garbage collection.** (Incorrect)

Question 8: **Incorrect**

Which of the following describes characteristics of the Dataset API?

- ○ **The Dataset API does not support unstructured data.**
- ○ **In Python, the Dataset API mainly resembles Pandas' DataFrame API.**
- ○ **In Python, the Dataset API's schema is constructed via type hints.**
- ○ **The Dataset API is available in Scala, but it is not available in Python.** (Correct)
- ◉ **The Dataset API does not provide compile-time type safety.** (Incorrect)

Question 9: **Correct**

Which of the following describes the difference between client and cluster execution modes?

- (●) **In cluster mode, the driver runs on the worker nodes, while the client mode runs the driver on the client machine.** (Correct)
- ( ) **In cluster mode, the driver runs on the edge node, while the client mode runs the driver in a worker node.**
- ( ) **In cluster mode, each node will launch its own executor, while in client mode, executors will exclusively run on the client machine.**
- ( ) **In client mode, the cluster manager runs on the same host as the driver, while in cluster mode, the cluster manager runs on a separate node.**
- ( ) **In cluster mode, the driver runs on the master node, while in client mode, the driver runs on a virtual machine in the cloud.**

Question 10: **Correct**

Which of the following statements about executors is correct, assuming that one can consider each of the JVMs working as executors as a pool of task execution slots?

- ( ) **Slot is another name for executor.**
- ( ) **There has to be a smaller number of executors than tasks.**
- ( ) **An executor runs on a single core.**
- ( ) **There has to be a greater number of slots than tasks.**
- (●) **Tasks run in parallel via slots.** (Correct)

Question 11: **Correct**

Which of the following statements about RDDs is incorrect?

- (●) **An RDD consists of a single partition.** (Correct)
- ( ) **The high-level DataFrame API is built on top of the low-level RDD API.**
- ( ) **RDDs are immutable.**
- ( ) **RDD stands for Resilient Distributed Dataset.**
- ( ) **RDDs are great for precisely instructing Spark on how to do a query.**

Question 12: **Incorrect**

Which of the elements that are labeled with a circle and a number contain an error or are misrepresented?

- ○ **1, 10**
- ○ **1, 8** **(Correct)**
- ○ **10**
- ◉ **7, 9, 10** **(Incorrect)**
- ○ **1, 4, 6, 9**

Question 13: Incorrect

Which of the following describes characteristics of the Spark UI?

- (●) **Via the Spark UI, workloads can be manually distributed across executors.** (Incorrect)
- ( ) **Via the Spark UI, stage execution speed can be modified.**
- ( ) **The Scheduler tab shows how jobs that are run in parallel by multiple users are distributed across the cluster.**
- ( ) **There is a place in the Spark UI that shows the property `spark.executor.memory`.** (Correct)
- ( ) **Some of the tabs in the Spark UI are named Jobs, Stages, Storage, DAGs, Executors, and SQL.**

Question 14: Incorrect

Which of the following statements about broadcast variables is correct?

- ( ) **Broadcast variables are serialized with every single task.**
- ( ) **Broadcast variables are commonly used for tables that do not fit into memory.**
- ( ) **Broadcast variables are immutable.** (Correct)
- ( ) **Broadcast variables are occasionally dynamically updated on a per-task basis.**
- (●) **Broadcast variables are local to the worker node and not shared across the cluster.** (Incorrect)

Question 15: Incorrect

Which of the following is a viable way to improve Spark's performance when dealing with large amounts of data, given that there is only a single application running on the cluster?

- ○ **Increase values for the properties** `spark.default.parallelism` **and** `spark.sql.shuffle.partitions` (Correct)
- ○ **Decrease values for the properties** `spark.default.parallelism` **and** `spark.sql.partitions`
- ◉ **Increase values for the properties** `spark.sql.parallelism` **and** `spark.sql.partitions` (Incorrect)
- ○ **Increase values for the properties** `spark.sql.parallelism` **and** `spark.sql.shuffle.partitions`
- ○ **Increase values for the properties** `spark.dynamicAllocation.maxExecutors`, `spark.default.parallelism`, **and** `spark.sql.shuffle.partitions`

Question 16: Correct

Which of the following describes a shuffle?

- ○ **A shuffle is a process that is executed during a broadcast hash join.**
- ○ **A shuffle is a process that compares data across executors.**
- ◉ **A shuffle is a process that compares data between partitions.** (Correct)
- ○ **A shuffle is a Spark operation that results from** `DataFrame.coalesce()`.
- ○ **A shuffle is a process that allocates partitions to executors.**

Question 17: **Incorrect**

Which of the following describes Spark's Adaptive Query Execution?

- ○ **Adaptive Query Execution features include dynamically coalescing shuffle partitions, dynamically injecting scan filters, and dynamically optimizing skew joins.**
- ○ **Adaptive Query Execution is enabled in Spark by default.**
- ◉ **Adaptive Query Execution reoptimizes queries at execution points.** (Incorrect)
- ○ **Adaptive Query Execution features are dynamically switching join strategies and dynamically optimizing skew joins.** (Correct)
- ○ **Adaptive Query Execution applies to all kinds of queries.**

Question 18: **Correct**

The code block displayed below contains an error. The code block is intended to join DataFrame `itemsDf` with the larger DataFrame `transactionsDf` on column `itemId`. Find the error.

Code block:

```
transactionsDf.join(itemsDf, "itemId", how="broadcast")
```

- ○ **The syntax is wrong, `how=` should be removed from the code block.**
- ○ **The `join` method should be replaced by the `broadcast` method.**
- ○ **Spark will only perform the `broadcast` operation if this behavior has been enabled on the Spark cluster.**
- ○ **The larger DataFrame `transactionsDf` is being broadcasted, rather than the smaller DataFrame `itemsDf`.**
- ◉ **`broadcast` is not a valid join type.** (Correct)

Question 19: **Correct**

Which of the following code blocks efficiently converts DataFrame `transactionsDf` from 12 into 24 partitions?

- ○ `transactionsDf.repartition(24, boost=True)`
- ○ `transactionsDf.repartition()`
- ○ `transactionsDf.repartition("itemId", 24)`
- ○ `transactionsDf.coalesce(24)`
- ◉ `transactionsDf.repartition(24)` **(Correct)**

Question 20: **Incorrect**

Which of the following code blocks removes all rows in the 6-column DataFrame `transactionsDf` that have missing data in at least 3 columns?

- ○ `transactionsDf.dropna("any")`
- ○ `transactionsDf.dropna(thresh=4)` **(Correct)**
- ○ `transactionsDf.drop.na("",2)`
- ◉ `transactionsDf.dropna(thresh=2)` **(Incorrect)**
- ○ `transactionsDf.dropna("",4)`

Question 21: **Correct**

Which of the following code blocks can be used to save DataFrame `transactionsDf` to memory only, recalculating partitions that do not fit in memory when they are needed?

○
```
1 from pyspark import StorageLevel
2 transactionsDf.cache(StorageLevel.MEMORY_ONLY)
```

○ `transactionsDf.cache()`

○ `transactionsDf.storage_level('MEMORY_ONLY')`

○ `transactionsDf.persist()`

○ `transactionsDf.clear_persist()`

◉
```
1 from pyspark import StorageLevel
2 transactionsDf.persist(StorageLevel.MEMORY_ONLY)
```
(Correct)

Explanation

```
1 from pyspark import StorageLevel
2 transactionsDf.persist(StorageLevel.MEMORY_ONLY)
```

Correct. Note that the storage level `MEMORY_ONLY` means that all partitions that do not fit into memory will be recomputed when they are needed.

Question 22: **Correct**

The code block displayed below contains an error. The code block should create DataFrame `itemsAttributesDf` which has columns `itemId` and `attribute` and lists every attribute from the `attributes` column in DataFrame `itemsDf` next to the `itemId` of the respective row in `itemsDf`. Find the error.

A sample of DataFrame `itemsDf` is below.

```
+------+------------------------------+------------------+
|itemId|attributes                    |supplier          |
+------+------------------------------+------------------+
|1     |[blue, winter, cozy]          |Sports Company Inc.|
|2     |[red, summer, fresh, cooling]|YetiX             |
|3     |[green, summer, travel]       |Sports Company Inc.|
+------+------------------------------+------------------+
```

Code block:

```
itemsAttributesDf =
itemsDf.explode("attributes").alias("attribute").select("attribute",
"itemId")
```

- ○ **Since `itemId` is the index, it does not need to be an argument to the `select()` method.**
- ○ **The `alias()` method needs to be called after the `select()` method.**
- ○ **The `explode()` method expects a `Column` object rather than a string.**
- ◉ **`explode()` is not a method of `DataFrame`. `explode()` should be used inside the `select()` method instead.** (Correct)
- ○ **The `split()` method should be used inside the `select()` method instead of the `explode()` method.**

Question 23: **Correct**

Which of the following code blocks reads in parquet file `/FileStore/imports.parquet` as a DataFrame?

- ○ `spark.mode("parquet").read("/FileStore/imports.parquet")`
- ○ `spark.read.path("/FileStore/imports.parquet", source="parquet")`
- ○ `spark.read().parquet("/FileStore/imports.parquet")`
- ◉ `spark.read.parquet("/FileStore/imports.parquet")` (Correct)
- ○ `spark.read().format('parquet').open("/FileStore/imports.parquet")`

Question 24: **Correct**

The code block shown below should convert up to 5 rows in DataFrame `transactionsDf` that have the value 25 in column `storeId` into a Python list. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Code block:

```
transactionsDf.__1__(__2__).__3__(__4__)
```

- ○ 1. `filter` 2. `"storeId"==25` 3. `collect` 4. `5`
- ○ 1. `filter` 2. `col("storeId")==25` 3. `toLocalIterator` 4. `5`
- ○ 1. `select` 2. `storeId==25` 3. `head` 4. `5`
- ◉ 1. `filter` 2. `col("storeId")==25` 3. `take` 4. `5` **(Correct)**
- ○ 1. `filter` 2. `col("storeId")==25` 3. `collect` 4. `5`

Question 25: **Incorrect**

Which of the following code blocks reads JSON file `imports.json` into a DataFrame?

- ○ `spark.read().mode("json").path("/FileStore/imports.json")`
- ◉ `spark.read.format("json").path("/FileStore/imports.json")` **(Incorrect)**
- ○ `spark.read("json", "/FileStore/imports.json")`
- ○ `spark.read.json("/FileStore/imports.json")` **(Correct)**
- ○ `spark.read().json("/FileStore/imports.json")`

Question 26: **Correct**

Which of the following code blocks returns a DataFrame that has all columns of DataFrame `transactionsDf` and an additional column `predErrorSquared` which is the squared value of column `predError` in DataFrame `transactionsDf`?

- ○ `transactionsDf.withColumn("predError", pow(col("predErrorSquared"), 2))`
- ○ `transactionsDf.withColumnRenamed("predErrorSquared", pow(predError, 2))`
- ◉ `transactionsDf.withColumn("predErrorSquared", pow(col("predError"), lit(2)))` **(Correct)**
- ○ `transactionsDf.withColumn("predErrorSquared", pow(predError, lit(2)))`
- ○ `transactionsDf.withColumn("predErrorSquared", "predError"**2)`

Question 27: **Correct**

The code block displayed below contains an error. The code block should return a new DataFrame that only contains rows from DataFrame `transactionsDf` in which the value in column `predError` is at least 5. Find the error.

Code block:

```
transactionsDf.where("col(predError) >= 5")
```

- (●) **The argument to the `where` method should be `"predError >= 5"`.** (Correct)
- ( ) **Instead of `where()`, `filter()` should be used.**
- ( ) **The expression returns the original DataFrame `transactionsDf` and not a new DataFrame. To avoid this, the code block should be `transactionsDf.toNewDataFrame().where("col(predError) >= 5")`.**
- ( ) **The argument to the `where` method cannot be a string.**
- ( ) **Instead of `>=`, the SQL operator `GEQ` should be used.**

Question 28: **Incorrect**

Which of the following code blocks saves DataFrame `transactionsDf` in location `/FileStore/transactions.csv` as a CSV file and throws an error if a file already exists in the location?

- ( ) `transactionsDf.write.save("/FileStore/transactions.csv")`
- (●) `transactionsDf.write.format("csv").mode("error").path("/FileStore/transactions.csv")` (Incorrect)
- ( ) `transactionsDf.write.format("csv").mode("ignore").path("/FileStore/transactions.csv")`
- ( ) `transactionsDf.write("csv").mode("error").save("/FileStore/transactions.csv")`
- ( ) `transactionsDf.write.format("csv").mode("error").save("/FileStore/transactions.csv")` (Correct)

Question 29: **Correct**

The code block shown below should return a DataFrame with two columns, `itemId` and `col`. In this DataFrame, for each element in column `attributes` of DataFrame `itemDf` there should be a separate row in which the column `itemId` contains the associated `itemId` from DataFrame `itemsDf`. The new DataFrame should only contain rows for rows in DataFrame `itemsDf` in which the column `attributes` contains the element `cozy`.

A sample of DataFrame `itemsDf` is below.

```
+------+------------------------------+--------------------+
|itemId|attributes                    |supplier            |
+------+------------------------------+--------------------+
|1     |[blue, winter, cozy]          |Sports Company Inc.|
|2     |[red, summer, fresh, cooling]|YetiX               |
|3     |[green, summer, travel]       |Sports Company Inc.|
+------+------------------------------+--------------------+
```

Code block:

```
itemsDf.__1__(__2__).__3__(__4__, __5__(__6__))
```

- ○
  1. `filter`
  2. `array_contains("cozy")`
  3. `select`
  4. `"itemId"`
  5. `explode`
  6. `"attributes"`

- ○
  1. `where`
  2. `"array_contains(attributes, 'cozy')"`
  3. `select`
  4. `itemId`
  5. `explode`
  6. `attributes`

- ○
  1. `filter`
  2. `"array_contains(attributes, 'cozy')"`
  3. `select`
  4. `"itemId"`
  5. `map`
  6. `"attributes"`

- ○
  1. `filter`
  2. `"array_contains(attributes, cozy)"`
  3. `select`
  4. `"itemId"`
  5. `explode`
  6. `"attributes"`

- ◉
  1. `filter`
  2. `"array_contains(attributes, 'cozy')"`
  3. `select`
  4. `"itemId"`
  5. `explode`
  6. `"attributes"`

  (Correct)

Question 30: **Correct**

The code block displayed below contains an error. The code block should return the average of rows in column `value` grouped by unique `storeId`. Find the error.

Code block:

```
transactionsDf.agg("storeId").avg("value")
```

- ○ **Instead of `avg("value")`, `avg(col("value"))` should be used.**
- ○ **The `avg("value")` should be specified as a second argument to `agg()` instead of being appended to it.**
- ○ **All column names should be wrapped in `col()` operators.**
- ◉ **`agg` should be replaced by `groupBy`.** (Correct)
- ○ **`"storeId"` and `"value"` should be swapped.**

Question 31: **Correct**

Which of the following code blocks returns a copy of DataFrame `itemsDf` where the column `supplier` has been renamed to `manufacturer`?

- ○ `itemsDf.withColumn(["supplier", "manufacturer"])`
- ○ `itemsDf.withColumn("supplier").alias("manufacturer")`
- ◉ `itemsDf.withColumnRenamed("supplier", "manufacturer")` (Correct)
- ○ `itemsDf.withColumnRenamed(col("manufacturer"), col("supplier"))`
- ○ `itemsDf.withColumnsRenamed("supplier", "manufacturer")`

Question 32: **Incorrect**

Which of the following code blocks returns DataFrame `transactionsDf` sorted in descending order by column `predError`, showing missing values last?

- ○ `transactionsDf.sort(asc_nulls_last("predError"))`
- ◉ `transactionsDf.orderBy("predError").desc_nulls_last()` **(Incorrect)**
- ○ `transactionsDf.sort("predError", ascending=False)` **(Correct)**
- ○ `transactionsDf.desc_nulls_last("predError")`
- ○ `transactionsDf.orderBy("predError").asc_nulls_last()`

Explanation

`transactionsDf.sort("predError", ascending=False)`

Correct! When using `DataFrame.sort()` and setting `ascending=False`, the DataFrame will be sorted by the specified column in descending order, putting all missing values last. An alternative, although not listed as an answer here, would be `transactionsDf.sort(desc_nulls_last("predError"))`.

Question 33: **Correct**

The code block displayed below contains an error. The code block is intended to perform an outer join of DataFrames `transactionsDf` and `itemsDf` on columns `productId` and `itemId`, respectively. Find the error.

Code block:

```
transactionsDf.join(itemsDf, [itemsDf.itemId, transactionsDf.productId], "outer")
```

- ○ **The `"outer"` argument should be eliminated, since "outer" is the default join type.**
- ○ **The join type needs to be appended to the `join()` operator, like `join().outer()` instead of listing it as the last argument inside the `join()` call.**
- ◉ **The term `[itemsDf.itemId, transactionsDf.productId]` should be replaced by `itemsDf.itemId == transactionsDf.productId`.** **(Correct)**
- ○ **The term `[itemsDf.itemId, transactionsDf.productId]` should be replaced by `itemsDf.col("itemId") == transactionsDf.col("productId")`.**
- ○ **The `"outer"` argument should be eliminated from the call and `join` should be replaced by `joinOuter`.**

Question 34: **Correct**

Which of the following code blocks performs a join in which the small DataFrame `transactionsDf` is sent to all executors where it is joined with DataFrame `itemsDf` on columns `storeId` and `itemId`, respectively?

- ○ `itemsDf.join(transactionsDf, itemsDf.itemId == transactionsDf.storeId, "right_outer")`
- ○ `itemsDf.join(transactionsDf, itemsDf.itemId == transactionsDf.storeId, "broadcast")`
- ○ `itemsDf.merge(transactionsDf, "itemsDf.itemId == transactionsDf.storeId", "broadcast")`
- ◉ `itemsDf.join(broadcast(transactionsDf), itemsDf.itemId == transactionsDf.storeId)` **(Correct)**
- ○ `itemsDf.join(transactionsDf, broadcast(itemsDf.itemId == transactionsDf.storeId))`

Question 35: **Correct**

Which of the following code blocks reduces a DataFrame from 12 to 6 partitions and performs a full shuffle?

- ○ `DataFrame.repartition(12)`
- ○ `DataFrame.coalesce(6).shuffle()`
- ○ `DataFrame.coalesce(6)`
- ○ `DataFrame.coalesce(6, shuffle=True)`
- ◉ `DataFrame.repartition(6)` **(Correct)**

Question 36: **Correct**

The code block displayed below contains an error. The code block is intended to write DataFrame `transactionsDf` to disk as a parquet file in location `/FileStore/transactions_split`, using column `storeId` as key for partitioning. Find the error.

Code block:

```
transactionsDf.write.format("parquet").partitionOn("storeId").save("/FileStore/transactions_split")
```

- ○ **The `format("parquet")` expression is inappropriate to use here, `"parquet"` should be passed as first argument to the `save()` operator and `"/FileStore/transactions_split"` as the second argument.**
- ◉ **Partitioning data by `storeId` is possible with the `partitionBy` expression, so `partitionOn` should be replaced by `partitionBy`.** (Correct)
- ○ **Partitioning data by `storeId` is possible with the `bucketBy` expression, so `partitionOn` should be replaced by `bucketBy`.**
- ○ **`partitionOn("storeId")` should be called before the `write` operation.**
- ○ **The `format("parquet")` expression should be removed and instead, the information should be added to the `write` expression like so: `write("parquet")`.**

Question 37: **Correct**

The code block displayed below contains an error. The code block is intended to return all columns of DataFrame `transactionsDf` except for columns `predError`, `productId`, and `value`. Find the error.

Excerpt of DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|   f|
+-------------+---------+-----+-------+---------+----+
|            1|        3|    4|     25|        1|null|
|            2|        6|    7|      2|        2|null|
|            3|        3| null|     25|        3|null|
+-------------+---------+-----+-------+---------+----+
```

Code block:

```
transactionsDf.select(~col("predError"), ~col("productId"),
~col("value"))
```

- ○ The `select` operator should be replaced by the `drop` operator and the arguments to the `drop` operator should be column names `predError`, `productId` and `value` wrapped in the `col` operator so they should be expressed like `drop(col(predError), col(productId), col(value))`.
- ○ The `select` operator should be replaced with the `deselect` operator.
- ○ The column names in the `select` operator should not be strings and wrapped in the `col` operator, so they should be expressed like `select(~col(predError), ~col(productId), ~col(value))`.
- ○ The `select` operator should be replaced by the `drop` operator.
- ◉ The `select` operator should be replaced by the `drop` operator and the arguments to the `drop` operator should be column names `predError`, `productId` and `value` as strings. (Correct)

Question 38: **Correct**

The code block displayed below contains an error. The code block should return DataFrame `transactionsDf`, but with the column `storeId` renamed to `storeNumber`. Find the error.

Code block:

```
transactionsDf.withColumn("storeNumber", "storeId")
```

- ○ **Instead of `withColumn`, the `withColumnRenamed` method should be used.**
- ○ **Arguments `"storeNumber"` and `"storeId"` each need to be wrapped in a `col()` operator.**
- ○ **Argument `"storeId"` should be the first and argument `"storeNumber"` should be the second argument to the `withColumn` method.**
- ○ **The `withColumn` operator should be replaced with the `copyDataFrame` operator.**
- ◉ **Instead of `withColumn`, the `withColumnRenamed` method should be used and argument `"storeId"` should be the first and argument `"storeNumber"` should be the second argument to that method.** (Correct)

Question 39: **Correct**

Which of the following code blocks returns a DataFrame with an added column to DataFrame `transactionsDf` that shows the unix epoch timestamps in column `transactionDate` as strings in the format month/day/year in column `transactionDateFormatted`?

Excerpt of DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+---------------+
|transactionId|predError|value|storeId|productId|   f|transactionDate|
+-------------+---------+-----+-------+---------+----+---------------+
|            1|        3|    4|     25|        1|null|     1587915332|
|            2|        6|    7|      2|        2|null|     1586815312|
|            3|        3| null|     25|        3|null|     1585824821|
|            4|     null| null|      3|        2|null|     1583244275|
|            5|     null| null|   null|        2|null|     1575285427|
|            6|        3|    2|     25|        2|null|     1572733275|
+-------------+---------+-----+-------+---------+----+---------------+
```

- ○ `transactionsDf.withColumn("transactionDateFormatted", from_unixtime("transactionDate", format="dd/MM/yyyy"))`
- ○ `transactionsDf.withColumnRenamed("transactionDate", "transactionDateFormatted", from_unixtime("transactionDateFormatted", format="MM/dd/yyyy"))`
- ○ `transactionsDf.apply(from_unixtime(format="MM/dd/yyyy")).asColumn("transactionDateFormatted")`
- ◉ `transactionsDf.withColumn("transactionDateFormatted", from_unixtime("transactionDate", format="MM/dd/yyyy"))` **(Correct)**
- ○ `transactionsDf.withColumn("transactionDateFormatted", from_unixtime("transactionDate"))`

Question 40: **Incorrect**

The code block displayed below contains an error. When the code block below has executed, it should have divided DataFrame `transactionsDf` into 14 parts, based on columns `storeId` and `transactionDate` (in this order). Find the error.

Code block:

```
transactionsDf.coalesce(14, ("storeId", "transactionDate"))
```

- ○ **The parentheses around the column names need to be removed and `.select()` needs to be appended to the code block.**
- ○ **Operator `coalesce` needs to be replaced by `repartition`, the parentheses around the column names need to be removed, and `.count()` needs to be appended to the code block.** (Correct)
- ○ **Operator `coalesce` needs to be replaced by `repartition`, the parentheses around the column names need to be removed, and `.select()` needs to be appended to the code block.**
- ○ **Operator `coalesce` needs to be replaced by `repartition` and the parentheses around the column names need to be replaced by square brackets.**
- ◉ **Operator `coalesce` needs to be replaced by `repartition`.** (Incorrect)

Explanation

Correct code block:

```
transactionsDf.repartition(14, "storeId", "transactionDate").count()
```

Since we do not know how many partitions DataFrame `transactionsDf` has, we cannot safely use `coalesce`, since it would not make any change if the current number of partitions is smaller than 14. So, we need to use `repartition`.

In the Spark documentation, the call structure for `repartition` is shown like this: `DataFrame.repartition(numPartitions, *cols)`. The `*` operator means that any argument after `numPartitions` will be interpreted as column. Therefore, the brackets need to be removed.

Finally, the question specifies that after the execution the DataFrame should be divided. So, indirectly this question is asking us to append an action to the code block. Since `.select()` is a transformation. the only possible choice here is `.count()`.

Question 41: **Incorrect**

Which of the following code blocks creates a new DataFrame with two columns `season` and `wind_speed_ms` where column `season` is of data type `string` and column `wind_speed_ms` is of data type `double`?

- `spark.DataFrame({"season": ["winter","summer"], "wind_speed_ms": [4.5, 7.5]})`
- `spark.createDataFrame([("summer", 4.5), ("winter", 7.5)], ["season", "wind_speed_ms"])` **(Correct)**
- (selected)
```
from pyspark.sql import types as T
spark.createDataFrame((("summer", 4.5), ("winter", 7.5)),
T.StructType([T.StructField("season", T.CharType()),
T.StructField("season", T.DoubleType())]))
```
(Incorrect)
- `spark.newDataFrame([("summer", 4.5), ("winter", 7.5)], ["season", "wind_speed_ms"])`
- `spark.createDataFrame({"season": ["winter","summer"], "wind_speed_ms": [4.5, 7.5]})`

Question 42: **Correct**

The code block shown below should return a column that indicates through boolean variables whether rows in DataFrame `transactionsDf` have values greater or equal to 20 and smaller or equal to 30 in column `storeId` and have the value 2 in column `productId`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

```
transactionsDf.__1__((__2__.__3__) __4__ (__5__))
```

- ○
  1. `select`
  2. `col("storeId")`
  3. `between(20, 30)`
  4. `and`
  5. `col("productId")==2`

- ○
  1. `where`
  2. `col("storeId")`
  3. `geq(20).leq(30)`
  4. `&`
  5. `col("productId")==2`

- ○
  1. `select`
  2. `"storeId"`
  3. `between(20, 30)`
  4. `&&`
  5. `col("productId")==2`

- ○
  1. `select`
  2. `col("storeId")`
  3. `between(20, 30)`
  4. `&&`
  5. `col("productId")=2`

- ◉
  1. `select`
  2. `col("storeId")`
  3. `between(20, 30)` **(Correct)**
  4. `&`
  5. `col("productId")==2`

Question 43: **Correct**

Which of the following code blocks displays the 10 rows with the smallest values of column `value` in DataFrame `transactionsDf` in a nicely formatted way?

- ○ `transactionsDf.sort(asc(value)).show(10)`
- ◉ `transactionsDf.sort(col("value")).show(10)` **(Correct)**
- ○ `transactionsDf.sort(col("value").desc()).head()`
- ○ `transactionsDf.sort(col("value").asc()).print(10)`
- ○ `transactionsDf.orderBy("value").asc().show(10)`

Question 44: **Correct**

Which of the following code blocks uses a schema `fileSchema` to read a parquet file at location `filePath` into a DataFrame?

- ◉ `spark.read.schema(fileSchema).format("parquet").load(filePath)` **(Correct)**
- ○ `spark.read.schema("fileSchema").format("parquet").load(filePath)`
- ○ `spark.read().schema(fileSchema).parquet(filePath)`
- ○ `spark.read().schema(fileSchema).format(parquet).load(filePath)`
- ○ `spark.read.schema(fileSchema).open(filePath)`

Question 45: **Correct**

Which of the following code blocks returns only rows from DataFrame `transactionsDf` in which values in column `productId` are unique?

- ○ `transactionsDf.distinct("productId")`
- ◉ `transactionsDf.dropDuplicates(subset=["productId"])` **(Correct)**
- ○ `transactionsDf.drop_duplicates(subset="productId")`
- ○ `transactionsDf.unique("productId")`
- ○ `transactionsDf.dropDuplicates(subset="productId")`

Question 36: **Correct**

The code block displayed below contains an error. The code block is intended to write DataFrame `transactionsDf` to disk as a parquet file in location `/FileStore/transactions_split`, using column `storeId` as key for partitioning. Find the error.

Code block:

```
transactionsDf.write.format("parquet").partitionOn("storeId").save("/FileStore/transactions_split")
```

- ○ **The `format("parquet")` expression is inappropriate to use here, `"parquet"` should be passed as first argument to the `save()` operator and `"/FileStore/transactions_split"` as the second argument.**
- ◉ **Partitioning data by `storeId` is possible with the `partitionBy` expression, so `partitionOn` should be replaced by `partitionBy`.** (Correct)
- ○ **Partitioning data by `storeId` is possible with the `bucketBy` expression, so `partitionOn` should be replaced by `bucketBy`.**
- ○ **`partitionOn("storeId")` should be called before the `write` operation.**
- ○ **The `format("parquet")` expression should be removed and instead, the information should be added to the `write` expression like so: `write("parquet")`.**

Question 46: **Correct**

The code block displayed below contains an error. The code block below is intended to add a column `itemNameElements` to DataFrame `itemsDf` that includes an array of all words in column `itemName`. Find the error.

Sample of DataFrame `itemsDf`:

```
+------+----------------------------------+------------------+
|itemId|itemName                          |supplier          |
+------+----------------------------------+------------------+
|1     |Thick Coat for Walking in the Snow|Sports Company Inc.|
|2     |Elegant Outdoors Summer Dress     |YetiX             |
|3     |Outdoors Backpack                 |Sports Company Inc.|
+------+----------------------------------+------------------+
```

Code block:

```
itemsDf.withColumnRenamed("itemNameElements", split("itemName"))
```

- ○ **All column names need to be wrapped in the `col()` operator.**
- ○ **Operator `withColumnRenamed` needs to be replaced with operator `withColumn` and a second argument `","` needs to be passed to the `split` method.**
- ○ **Operator `withColumnRenamed` needs to be replaced with operator `withColumn` and the `split` method needs to be replaced by the `splitString` method.**
- ◉ **Operator `withColumnRenamed` needs to be replaced with operator `withColumn` and a second argument `" "` needs to be passed to the `split` method.** (Correct)
- ○ **The expressions `"itemNameElements"` and `split("itemName")` need to be swapped.**

Question 47: **Correct**

The code block shown below should return all rows of DataFrame `itemsDf` that have at least 3 items in column `itemNameElements`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Example of DataFrame `itemsDf`:

```
+------+----------------------------------+-------------------+-----------
------------------------------+
|itemId|itemName                          |supplier
|itemNameElements                          |
+------+----------------------------------+-------------------+-----------
------------------------------+
|1     |Thick Coat for Walking in the Snow|Sports Company Inc.|[Thick,
Coat, for, Walking, in, the, Snow]|
|2     |Elegant Outdoors Summer Dress     |YetiX              |[Elegant,
Outdoors, Summer, Dress]         |
|3     |Outdoors Backpack                 |Sports Company Inc.|[Outdoors,
Backpack]                        |
+------+----------------------------------+-------------------+-----------
------------------------------+
```

Code block:

```
itemsDf.__1__(__2__(__3__)__4__)
```

- 1. `select`
  2. `count`
  3. `col("itemNameElements")`
  4. `>3`

- 1. `filter`
  2. `count`
  3. `itemNameElements`
  4. `>=3`

- 1. `select`
  2. `count`
  3. `"itemNameElements"`
  4. `>3`

- 1. `filter`
  2. `size`
  3. `"itemNameElements"`
  4. `>=3`

(Correct)

- 1. `select`
  2. `size`
  3. `"itemNameElements"`
  4. `>3`

Question 48: **Correct**

The code block displayed below contains an error. The code block should use Python method `find_most_freq_letter` to find the letter present most in column `itemName` of DataFrame `itemsDf` and return it in a new column `most_frequent_letter`. Find the error.

Code block:

```
1 find_most_freq_letter_udf = udf(find_most_freq_letter)
2 itemsDf.withColumn("most_frequent_letter",
find_most_freq_letter("itemName"))
```

- **Spark is not using the UDF method correctly.** (Correct)
- **The UDF method is not registered correctly, since the return type is missing.**
- **The `"itemName"` expression should be wrapped in `col()`.**
- **UDFs do not exist in PySpark.**
- **Spark is not adding a column.**

Question 49: **Incorrect**

Which of the following code blocks returns about 150 randomly selected rows from the 1000-row DataFrame `transactionsDf`, assuming that any row can appear more than once in the returned DataFrame?

- ○ `transactionsDf.resample(0.15, False, 3142)`
- ○ `transactionsDf.sample(0.15, False, 3142)`
- ◉ `transactionsDf.sample(0.15)` **(Incorrect)**
- ○ `transactionsDf.sample(0.85, 8429)`
- ○ `transactionsDf.sample(True, 0.15, 8261)` **(Correct)**

Explanation

Answering this question correctly depends on whether you understand the arguments to the `DataFrame.sample()` method (link to the documentation below). The arguments are as follows: `DataFrame.sample(withReplacement=None, fraction=None, seed=None)`.

The first argument `withReplacement` specified whether a row can be drawn from the DataFrame multiple times. By default, this option is disabled in Spark. But we have to enable it here, since the question asks for a row being able to appear more than once. So, we need to pass `True` for this argument.

Question 50: **Correct**

Which of the following code blocks returns a DataFrame where columns `predError` and `productId` are removed from DataFrame `transactionsDf`?

Sample of DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|f   |
+-------------+---------+-----+-------+---------+----+
|1            |3        |4    |25     |1        |null|
|2            |6        |7    |2      |2        |null|
|3            |3        |null |25     |3        |null|
+-------------+---------+-----+-------+---------+----+
```

- ○ `transactionsDf.withColumnRemoved("predError", "productId")`
- ○ `transactionsDf.drop(["predError", "productId", "associateId"])`
- ◉ `transactionsDf.drop("predError", "productId", "associateId")` (Correct)
- ○ `transactionsDf.dropColumns("predError", "productId", "associateId")`
- ○ `transactionsDf.drop(col("predError", "productId"))`

Question 51: **Correct**

The code block displayed below contains an error. The code block should return a DataFrame where all entries in column `supplier` contain the letter combination `et` in this order. Find the error.

Code block:

```
itemsDf.filter(Column('supplier').isin('et'))
```

- ◉ **The `Column` operator should be replaced by the `col` operator and instead of `isin`, `contains` should be used.** (Correct)
- ○ **The expression inside the filter parenthesis is malformed and should be replaced by `isin('et', 'supplier')`.**
- ○ **Instead of `isin`, it should be checked whether column `supplier` contains the letters `et`, so `isin` should be replaced with `contains`. In addition, the column should be accessed using `col['supplier']`.**
- ○ **The expression only returns a single column and `filter` should be replaced by `select`.**

Question 52: **Incorrect**

The code block shown below should write DataFrame `transactionsDf` to disk at path `csvPath` as a single CSV file, using tabs (`\t` characters) as separators between columns, expressing missing values as string `n/a`, and omitting a header row with column names. Choose the answer that correctly fills the blanks in the code block to accomplish this.

```
transactionsDf.__1__.write.__2__(__3__, "\t").__4__.__5__(csvPath)
```

○
1. `coalesce(1)`
2. `option`
3. `"sep"`
4. `option("header", True)`
5. `path`

○
1. `coalesce(1)`
2. `option`
3. `"colsep"`
4. `option("nullValue", "n/a")`
5. `path`

○
1. `repartition(1)`
2. `option`
3. `"sep"` (Correct)
4. `option("nullValue", "n/a")`
5. `csv`

◉
1. `csv`
2. `option`
3. `"sep"` (Incorrect)
4. `option("emptyValue", "n/a")`
5. `path`

○
1. `repartition(1)`
2. `mode`
3. `"sep"`
4. `mode("nullValue", "n/a")`
5. `csv`

Question 53: **Correct**

Which of the following code blocks returns a new DataFrame with only columns `predError` and `values` of every second row of DataFrame `transactionsDf`?

Entire DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|   f|
+-------------+---------+-----+-------+---------+----+
|            1|        3|    4|     25|        1|null|
|            2|        6|    7|      2|        2|null|
|            3|        3| null|     25|        3|null|
|            4|     null| null|      3|        2|null|
|            5|     null| null|   null|        2|null|
|            6|        3|    2|     25|        2|null|
+-------------+---------+-----+-------+---------+----+
```

- ○ `transactionsDf.filter(col("transactionId").isin([3,4,6])).select([predError, value])`
- ○ `transactionsDf.select(col("transactionId").isin([3,4,6]), "predError", "value")`
- ○ `transactionsDf.filter("transactionId" % 2 == 0).select("predError", "value")`
- ◉ `transactionsDf.filter(col("transactionId") % 2 == 0).select("predError", "value")` **(Correct)**
- ○

```
transactionsDf.createOrReplaceTempView("transactionsDf")
spark.sql("FROM transactionsDf SELECT predError, value WHERE transactionId % 2 = 2")
```

- ○ `transactionsDf.filter(col(transactionId).isin([3,4,6]))`

Question 54: **Correct**

The code block displayed below contains an error. The code block should display the schema of DataFrame `transactionsDf`. Find the error.

Code block:

```
transactionsDf.rdd.printSchema
```

- ○ **There is no way to print a schema directly in Spark, since the schema can be printed easily through using `print(transactionsDf.columns)`, so that should be used instead.**
- ○ **The code block should be wrapped into a `print()` operation.**
- ○ **`printSchema` is only accessible through the spark session, so the code block should be rewritten as `spark.printSchema(transactionsDf)`.**
- ◉ **`printSchema` is a method and should be written as `printSchema()`. It is also not callable through `transactionsDf.rdd`, but should be called directly from `transactionsDf`.** (Correct)
- ○ **`printSchema` is a not a method of `transactionsDf.rdd`. Instead, the schema should be printed via `transactionsDf.print_schema()`.**

Question 55: **Correct**

Which of the following code blocks returns a one-column DataFrame of all values in column `supplier` of DataFrame `itemsDf` that do not contain the letter `X`? In the DataFrame, every value should only be listed once.

Sample of DataFrame `itemsDf`:

```
+------+--------------------+--------------------+-------------------+
|itemId|            itemName|          attributes|           supplier|
+------+--------------------+--------------------+-------------------+
|     1|Thick Coat for Wa...|[blue, winter, cozy]|Sports Company Inc.|
|     2|Elegant Outdoors ...|[red, summer, fre...|               YetiX|
|     3|   Outdoors Backpack|[green, summer, t...|Sports Company Inc.|
+------+--------------------+--------------------+-------------------+
```

- ○ `itemsDf.filter(col(supplier).not_contains('X')).select(supplier).distinct()`
- ○ `itemsDf.select(~col('supplier').contains('X')).distinct()`
- ○ `itemsDf.filter(not(col('supplier').contains('X'))).select('supplier').unique()`
- ◉ `itemsDf.filter(~col('supplier').contains('X')).select('supplier').distinct()` **(Correct)**
- ○ `itemsDf.filter(!col('supplier').contains('X')).select(col('supplier')).unique()`

Question 56: **Correct**

In which order should the code blocks shown below be run in order to create a table of all values in column `attributes` next to the respective values in column `supplier` in DataFrame `itemsDf`?

1. `itemsDf.createOrReplaceView("itemsDf")`
2. `spark.sql("SELECT 'supplier', explode('Attributes') FROM itemsDf")`
3. `spark.sql("SELECT supplier, explode(attributes) FROM itemsDf")`
4. `itemsDf.createOrReplaceTempView("itemsDf")`

- ◉ **4, 3** **(Correct)**
- ○ **1, 3**
- ○ **2**
- ○ **4, 2**
- ○ **1, 2**

Question 57: **Correct**

The code block shown below should return a copy of DataFrame `transactionsDf` without columns `value` and `productId` and with an additional column `associateId` that has the value `5`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

```
transactionsDf.__1__(__2__, __3__).__4__(__5__, 'value')
```

- 1. `withColumn`
  2. `'associateId'`
  3. `5`
  4. `remove`
  5. `'productId'`

- 1. `withNewColumn`
  2. `associateId`
  3. `lit(5)`
  4. `drop`
  5. `productId`

- **1. `withColumn`**
  2. `'associateId'`
  3. `lit(5)` **(Correct)**
  4. `drop`
  5. `'productId'`

- 1. `withColumnRenamed`
  2. `'associateId'`
  3. `5`
  4. `drop`
  5. `'productId'`

- 1. `withColumn`
  2. `col(associateId)`
  3. `lit(5)`
  4. `drop`
  5. `col(productId)`

Question 58: **Incorrect**

The code block shown below should write DataFrame `transactionsDf` as a parquet file to path `storeDir`, using brotli compression and replacing any previously existing file. Choose the answer that correctly fills the blanks in the code block to accomplish this.

```
transactionsDf.__1__.format("parquet").__2__(__3__).option(__4__, "brotli").__5__(storeDir)
```

- ○ 1. `save` 2. `mode` 3. `"ignore"` 4. `"compression"` 5. `path`
- ○ 1. `store` 2. `with` 3. `"replacement"` 4. `"compression"` 5. `path`
- ○ 1. `write` 2. `mode` 3. `"overwrite"` 4. `"compression"` 5. `save` **(Correct)**
- ○ 1. `save` 2. `mode` 3. `"replace"` 4. `"compression"` 5. `path`
- ◉ 1. `write` 2. `mode` 3. `"overwrite"` 4. `compression` 5. `parquet` **(Incorrect)**

Question 59: **Incorrect**

Which of the following code blocks will silently skip writing `itemsDf` in avro format to location `fileLocation` if there is a file at that location already?

- ○ `itemsDf.write.avro(fileLocation)`
- ○ `itemsDf.write.format("avro").mode("ignore").save(fileLocation)` **(Correct)**
- ○ `itemsDf.write.format("avro").mode("errorifexists").save(fileLocation)`
- ◉ `itemsDf.save.format("avro").mode("ignore").write(fileLocation)` **(Incorrect)**
- ○ `spark.DataFrameWriter(itemsDf).format("avro").write(fileLocation)`

Question 60: **Correct**

The code block displayed below contains at least one error. The code block should return a DataFrame with only one column, `result`. That column should include all values in column `value` from DataFrame `transactionsDf` raised to the power of 5, and a `null` value for rows in which there is no value in column `value`. Find the error(s).

Code block:

```
1  from pyspark.sql.functions import udf
2  from pyspark.sql import types as T
3
4  transactionsDf.createOrReplaceTempView('transactions')
5
6  def pow_5(x):
7    return x**5
8
9  spark.udf.register(pow_5, 'power_5_udf', T.LongType())
10 spark.sql('SELECT power_5_udf(value) FROM transactions')
```

- ○ **The `pow_5` method is unable to handle empty values in column `value` and the name of the column in the returned DataFrame is not `result`.**
- ○ **The returned DataFrame includes multiple columns instead of just one column.**
- ○ **The `pow_5` method is unable to handle empty values in column `value`, the name of the column in the returned DataFrame is not `result`, and the SparkSession cannot access the `transactionsDf` DataFrame.**
- ○ **The `pow_5` method is unable to handle empty values in column `value`, the name of the column in the returned DataFrame is not `result`, and Spark driver does not call the UDF function appropriately.**
- ◉ **The `pow_5` method is unable to handle empty values in column `value`, the UDF function is not registered properly with the Spark driver, and the name of the column in the returned DataFrame is not `result`.** (Correct)

Practice Test (Python) - 2

Question 1: **Correct**

Which of the following statements about Spark's execution hierarchy is correct?

- (●) **In Spark's execution hierarchy, a job may reach over multiple stage boundaries.** (Correct)
- ( ) **In Spark's execution hierarchy, manifests are one layer above jobs.**
- ( ) **In Spark's execution hierarchy, a stage comprises multiple jobs.**
- ( ) **In Spark's execution hierarchy, executors are the smallest unit.**
- ( ) **In Spark's execution hierarchy, tasks are one layer above slots.**

Question 2: **Incorrect**

Which of the following describes slots?

- (●) **Slots are dynamically created and destroyed in accordance with an executor's workload.** (Incorrect)
- ( ) **To optimize I/O performance, Spark stores data on disk in multiple slots.**
- ( ) **A Java Virtual Machine (JVM) working as an executor can be considered as a pool of slots for task execution.** (Correct)
- ( ) **A slot is always limited to a single core.**
- ( ) **Slots are the communication interface for executors and are used for receiving commands and sending results to the driver.**

Question 3: **Correct**

Which of the following describes the conversion of a computational query into an execution plan in Spark?

- ○ **Spark uses the catalog to resolve the optimized logical plan.**
- ○ **The catalog assigns specific resources to the optimized memory plan.**
- ◉ **The executed physical plan depends on a cost optimization from a previous stage.** (Correct)
- ○ **Depending on whether DataFrame API or SQL API are used, the physical plan may differ.**
- ○ **The catalog assigns specific resources to the physical plan.**

Question 4: **Correct**

Which of the following describes characteristics of the Spark driver?

- ○ **The Spark driver requests the transformation of operations into DAG computations from the worker nodes.**
- ○ **If set in the Spark configuration, Spark scales the Spark driver horizontally to improve parallel processing performance.**
- ○ **The Spark driver processes partitions in an optimized, distributed fashion.**
- ○ **In a non-interactive Spark application, the Spark driver automatically creates the `SparkSession` object.**
- ◉ **The Spark driver's responsibility includes scheduling queries for execution on worker nodes.** (Correct)

Question 5: **Correct**

Which of the following describes executors?

- ◯ **Executors host the Spark driver on a worker-node basis.**
- ◉ **Executors are responsible for carrying out work that they get assigned by the driver.** (Correct)
- ◯ **After the start of the Spark application, executors are launched on a per-task basis.**
- ◯ **Executors are located in slots inside worker nodes.**
- ◯ **The executors' storage is ephemeral and as such it defers the task of caching data directly to the worker node thread.**

Question 6: **Correct**

Which of the following statements about DAGs is correct?

- ◯ **DAGs help direct how Spark executors process tasks, but are a limitation to the proper execution of a query when an executor fails.**
- ◯ **DAG stands for "Directing Acyclic Graph".**
- ◯ **Spark strategically hides DAGs from developers, since the high degree of automation in Spark means that developers never need to consider DAG layouts.**
- ◯ **In contrast to transformations, DAGs are never lazily executed.**
- ◉ **DAGs can be decomposed into tasks that are executed in parallel.** (Correct)

Question 7: **Correct**

Which of the following DataFrame methods is classified as a transformation?

- ○ `DataFrame.count()`
- ○ `DataFrame.show()`
- ◉ `DataFrame.select()` (Correct)
- ○ `DataFrame.foreach()`
- ○ `DataFrame.first()`

Question 8: **Correct**

Which of the following statements about lazy evaluation is incorrect?

- ○ **Predicate pushdown is a feature resulting from lazy evaluation.**
- ◉ **Execution is triggered by transformations.** (Correct)
- ○ **Spark will fail a job only during execution, but not during definition.**
- ○ **Accumulators do not change the lazy evaluation model of Spark.**
- ○ **Lineages allow Spark to coalesce transformations into stages.**

Question 9: **Correct**

Which of the following describes how Spark achieves fault tolerance?

- ○ **Spark helps fast recovery of data in case of a worker fault by providing the `MEMORY_AND_DISK` storage level option.**
- ◉ **If an executor on a worker node fails while calculating an RDD, that RDD can be recomputed by another executor using the lineage.** (Correct)
- ○ **Spark builds a fault-tolerant layer on top of the legacy RDD data system, which by itself is not fault tolerant.**
- ○ **Due to the mutability of DataFrames after transformations, Spark reproduces them using observed lineage in case of worker node failure.**
- ○ **Spark is only fault-tolerant if this feature is specifically enabled via the `spark.fault_recovery.enabled` property.**

Question 10: **Correct**

Which is the highest level in Spark's execution hierarchy?

- ○ **Task**
- ○ **Executor**
- ○ **Slot**
- ◉ **Job** (Correct)
- ○ **Stage**

Question 11: **Correct**

Which of the following describes Spark's way of managing memory?

- ◯ **Spark uses a subset of the reserved system memory.**
- ◉ **Storage memory is used for caching partitions derived from DataFrames.** (Correct)
- ◯ **As a general rule for garbage collection, Spark performs better on many small objects than few big objects.**
- ◯ **Disabling serialization potentially greatly reduces the memory footprint of a Spark application.**
- ◯ **Spark's memory usage can be divided into three categories: Execution, transaction, and storage.**

Question 12: **Correct**

Which of the following statements about Spark's DataFrames is incorrect?

- ◯ **Spark's DataFrames are immutable.**
- ◉ **Spark's DataFrames are equal to Python's DataFrames.** (Correct)
- ◯ **Data in DataFrames is organized into named columns.**
- ◯ **RDDs are at the core of DataFrames.**
- ◯ **The data in DataFrames may be split into multiple chunks.**

Question 13: Incorrect

Which of the following statements about Spark's configuration properties is incorrect?

- (●) **The maximum number of tasks that an executor can process at the same time is controlled by the `spark.task.cpus` property.** (Incorrect)
- ( ) **The maximum number of tasks that an executor can process at the same time is controlled by the `spark.executor.cores` property.**
- ( ) **The default value for `spark.sql.autoBroadcastJoinThreshold` is 10MB.**
- ( ) **The default number of partitions to use when shuffling data for joins or aggregations is 300.** (Correct)
- ( ) **The default number of partitions returned from certain transformations can be controlled by the `spark.default.parallelism` property.**

Question 14: Incorrect

Which of the following describes a way for resizing a DataFrame from 16 to 8 partitions in the most efficient way?

- ( ) **Use operation `DataFrame.repartition(8)` to shuffle the DataFrame and reduce the number of partitions.**
- (●) **Use operation `DataFrame.coalesce(8)` to fully shuffle the DataFrame and reduce the number of partitions.** (Incorrect)
- ( ) **Use a narrow transformation to reduce the number of partitions.** (Correct)
- ( ) **Use a wide transformation to reduce the number of partitions.**
- ( ) **Use operation `DataFrame.coalesce(0.5)` to halve the number of partitions in the DataFrame.**

Question 15: **Correct**

Which of the following DataFrame operators is never classified as a wide transformation?

- ○ `DataFrame.sort()`
- ○ `DataFrame.aggregate()`
- ○ `DataFrame.repartition()`
- ◉ `DataFrame.select()` (Correct)
- ○ `DataFrame.join()`

Question 16: **Correct**

Which of the following statements about data skew is incorrect?

- ○ **Spark will not automatically optimize skew joins by default.**
- ○ **Broadcast joins are a viable way to increase join performance for skewed data over sort-merge joins.**
- ○ **In skewed DataFrames, the largest and the smallest partition consume very different amounts of memory.**
- ◉ **To mitigate skew, Spark automatically disregards null values in keys when joining.** (Correct)
- ○ **Salting can resolve data skew.**

Question 17: **Incorrect**

Which of the following describes the characteristics of accumulators?

- ◯ **Accumulators are used to pass around lookup tables across the cluster.**
- ◯ **All accumulators used in a Spark application are listed in the Spark UI.**
- ◉ **Accumulators can be instantiated directly via the `accumulator(n)` method of the `pyspark.RDD` module.** (Incorrect)
- ◯ **Accumulators are immutable.**
- ◯ **If an action including an accumulator fails during execution and Spark manages to restart the action and complete it successfully, only the successful attempt will be counted in the accumulator.** (Correct)

Question 18: **Correct**

Which of the following code blocks returns a DataFrame with a single column in which all items in column `attributes` of DataFrame `itemsDf` are listed that contain the letter `i`?

Sample of DataFrame `itemsDf`:

```
+------+----------------------------------+----------------------------+-----------------+
|itemId|itemName                          |attributes                  |supplier         |
+------+----------------------------------+----------------------------+-----------------+
|1     |Thick Coat for Walking in the Snow|[blue, winter, cozy]        |Sports Company Inc.|
|2     |Elegant Outdoors Summer Dress     |[red, summer, fresh, cooling]|YetiX            |
|3     |Outdoors Backpack                 |[green, summer, travel]     |Sports Company Inc.|
+------+----------------------------------+----------------------------+-----------------+
```

- ○ `itemsDf.select(explode("attributes").alias("attributes_exploded")).filter(attributes_exploded.contains("i"))`
- ○ `itemsDf.explode(attributes).alias("attributes_exploded").filter(col("attributes_exploded").contains("i"))`
- ○ `itemsDf.select(explode("attributes")).filter("attributes_exploded".contains("i"))`
- ◉ `itemsDf.select(explode("attributes").alias("attributes_exploded")).filter(col("attributes_exploded").contains("i"))` **(Correct)**
- ○ `itemsDf.select(col("attributes").explode().alias("attributes_exploded")).filter(col("attributes_exploded").contains("i"))`

Question 19: **Correct**

Which of the following code blocks returns all unique values of column `storeId` in DataFrame `transactionsDf`?

- ○ `transactionsDf["storeId"].distinct()`
- ◉ `transactionsDf.select("storeId").distinct()` **(Correct)**
- ○ `transactionsDf.filter("storeId").distinct()`
- ○ `transactionsDf.select(col("storeId").distinct())`
- ○ `transactionsDf.distinct("storeId")`

Question 20: **Correct**

Which of the following code blocks stores a part of the data in DataFrame `itemsDf` on executors?

- ◉ `itemsDf.cache().count()` **(Correct)**
- ○ `itemsDf.cache(eager=True)`
- ○ `cache(itemsDf)`
- ○ `itemsDf.cache().filter()`
- ○ `itemsDf.rdd.storeCopy()`

Question 21: **Correct**

Which of the following code blocks selects all rows from DataFrame `transactionsDf` in which column `productId` is zero or smaller or equal to 3?

- ○ `transactionsDf.filter(productId==3 or productId<1)`
- ○ `transactionsDf.filter((col("productId")==3) or (col("productId")<1))`
- ○ `transactionsDf.filter(col("productId")==3 | col("productId")<1)`
- ○ `transactionsDf.where("productId"=3).or("productId"<1))`
- ◉ `transactionsDf.filter((col("productId")==3) | (col("productId")<1))` **(Correct)**

Question 22: **Correct**

Which of the following code blocks sorts DataFrame `transactionsDf` both by column `storeId` in ascending and by column `productId` in descending order, in this priority?

- ○ `transactionsDf.sort("storeId", asc("productId"))`
- ○ `transactionsDf.sort(col(storeId)).desc(col(productId))`
- ○ `transactionsDf.order_by(col(storeId), desc(col(productId)))`
- ◉ `transactionsDf.sort("storeId", desc("productId"))` **(Correct)**
- ○ `transactionsDf.sort("storeId").sort(desc("productId"))`

Question 23: **Correct**

The code block displayed below contains an error. The code block should produce a DataFrame with `color` as the only column and three rows with `color` values of `red`, `blue`, and `green`, respectively. Find the error.

Code block:

```
spark.createDataFrame([("red",), ("blue",), ("green",)], "color")
```

- ○ **Instead of calling `spark.createDataFrame`, just `DataFrame` should be called.**
- ○ **The commas in the tuples with the colors should be eliminated.**
- ○ **The colors `red`, `blue`, and `green` should be expressed as a simple Python list, and not a list of tuples.**
- ○ **Instead of `color`, a data type should be specified.**
- ◉ **The `"color"` expression needs to be wrapped in brackets, so it reads `["color"]`.** (Correct)

Question 24: **Correct**

The code block displayed below contains an error. The code block should return all rows of DataFrame `transactionsDf`, but including only columns `storeId` and `predError`. Find the error.

Code block:

```
spark.collect(transactionsDf.select("storeId", "predError"))
```

- ○ **Instead of `select`, DataFrame `transactionsDf` needs to be filtered using the `filter` operator.**
- ○ **Columns `storeId` and `predError` need to be represented as a Python list, so they need to be wrapped in brackets (`[]`).**
- ○ **The `take` method should be used instead of the `collect` method.**
- ○ **Instead of `collect`, `collectAsRows` needs to be called.**
- ◉ **The `collect` method is not a method of the `SparkSession` object.** (Correct)

Question 25: **Correct**

The code block shown below should store DataFrame `transactionsDf` on two different executors, utilizing the executors' memory as much as possible, but not writing anything to disk. Choose the answer that correctly fills the blanks in the code block to accomplish this.

```
from pyspark import StorageLevel
transactionsDf.__1__(StorageLevel.__2__).__3__
```

- ◯ 1. `cache` 2. `MEMORY_ONLY_2` 3. `count()`
- ◯ 1. `persist` 2. `DISK_ONLY_2` 3. `count()`
- ◯ 1. `persist` 2. `MEMORY_ONLY_2` 3. `select()`
- ◯ 1. `cache` 2. `DISK_ONLY_2` 3. `count()`
- ◉ 1. `persist` 2. `MEMORY_ONLY_2` 3. `count()` **(Correct)**

Question 26: **Correct**

The code block displayed below contains an error. The code block should return a copy of DataFrame `transactionsDf` where the name of column `transactionId` has been changed to `transactionNumber`. Find the error.

Code block:

```
transactionsDf.withColumn("transactionNumber", "transactionId")
```

- ○ **The arguments to the `withColumn` method need to be reordered.**
- ○ **The arguments to the `withColumn` method need to be reordered and the `copy()` operator should be appended to the code block to ensure a copy is returned.**
- ○ **The `copy()` operator should be appended to the code block to ensure a copy is returned.**
- ○ **Each column name needs to be wrapped in the `col()` method and method `withColumn` should be replaced by method `withColumnRenamed`.**
- ◉ **The method `withColumn` should be replaced by method `withColumnRenamed` and the arguments to the method need to be reordered.** (Correct)

Question 27: **Correct**

Which of the following code blocks performs an inner join between DataFrame `itemsDf` and DataFrame `transactionsDf`, using columns `itemId` and `transactionId` as join keys, respectively?

- ○ `itemsDf.join(transactionsDf, "inner", itemsDf.itemId == transactionsDf.transactionId)`
- ○ `itemsDf.join(transactionsDf, itemId == transactionId)`
- ◉ `itemsDf.join(transactionsDf, itemsDf.itemId == transactionsDf.transactionId, "inner")` (Correct)
- ○ `itemsDf.join(transactionsDf, "itemsDf.itemId == transactionsDf.transactionId", "inner")`
- ○ `itemsDf.join(transactionsDf, col(itemsDf.itemId) == col(transactionsDf.transactionId))`

Question 28: **Correct**

The code block shown below should return only the average prediction error (column `predError`) of a random subset, without replacement, of approximately 15% of rows in DataFrame `transactionsDf`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

```
transactionsDf.__1__(__2__, __3__).__4__(avg('predError'))
```

- 1. `sample`
  2. `True`
  3. `0.15`
  4. `filter`

- 1. `sample`
  2. `False`
  3. `0.15`
  4. `select`

(Correct)

- 1. `sample`
  2. `0.85`
  3. `False`
  4. `select`

- 1. `fraction`
  2. `0.15`
  3. `True`
  4. `where`

- 1. `fraction`
  2. `False`
  3. `0.85`
  4. `select`

Question 29: **Correct**

Which of the following code blocks returns a single-column DataFrame showing the number of words in column `supplier` of DataFrame `itemsDf`?

Sample of DataFrame `itemsDf`:

```
+------+----------------------------+------------------+
|itemId|attributes                  |supplier          |
+------+----------------------------+------------------+
|1     |[blue, winter, cozy]        |Sports Company Inc.|
|2     |[red, summer, fresh, cooling]|YetiX             |
|3     |[green, summer, travel]     |Sports Company Inc.|
+------+----------------------------+------------------+
```

- ○ `itemsDf.split("supplier", " ").count()`
- ○ `itemsDf.split("supplier", " ").size()`
- ○ `itemsDf.select(word_count("supplier"))`
- ○ `spark.select(size(split(col(supplier), " ")))`
- ◉ `itemsDf.select(size(split("supplier", " ")))` **(Correct)**

Question 30: **Correct**

Which of the following code blocks stores DataFrame `itemsDf` in executor memory and, if insufficient memory is available, serializes it and saves it to disk?

- ○ `itemsDf.persist(StorageLevel.MEMORY_ONLY)`
- ○ `itemsDf.cache(StorageLevel.MEMORY_AND_DISK)`
- ○ `itemsDf.store()`
- ◉ `itemsDf.cache()` **(Correct)**
- ○ `itemsDf.write.option('destination', 'memory').save()`

Question 31: **Correct**

Which of the following code blocks adds a column `predErrorSqrt` to DataFrame `transactionsDf` that is the square root of column `predError`?

- ○ `transactionsDf.withColumn("predErrorSqrt", sqrt(predError))`
- ○ `transactionsDf.select(sqrt(predError))`
- ○ `transactionsDf.withColumn("predErrorSqrt", col("predError").sqrt())`
- ◉ `transactionsDf.withColumn("predErrorSqrt", sqrt(col("predError")))` **(Correct)**
- ○ `transactionsDf.select(sqrt("predError"))`

Question 32: **Correct**

Which of the following code blocks reorders the values inside the arrays in column `attributes` of DataFrame `itemsDf` from last to first one in the alphabet?

```
+------+-----------------------------+-------------------+
|itemId|attributes                   |supplier           |
+------+-----------------------------+-------------------+
|1     |[blue, winter, cozy]         |Sports Company Inc.|
|2     |[red, summer, fresh, cooling]|YetiX              |
|3     |[green, summer, travel]      |Sports Company Inc.|
+------+-----------------------------+-------------------+
```

- ○ `itemsDf.withColumn('attributes', sort_array(col('attributes').desc()))`
- ○ `itemsDf.withColumn('attributes', sort_array(desc('attributes')))`
- ○ `itemsDf.withColumn('attributes', sort(col('attributes'), asc=False))`
- ◉ `itemsDf.withColumn("attributes", sort_array("attributes", asc=False))` **(Correct)**
- ○ `itemsDf.select(sort_array("attributes"))`

Question 33: **Correct**

Which of the following code blocks returns a copy of DataFrame `transactionsDf` where the column `storeId` has been converted to string type?

- ○ `transactionsDf.withColumn("storeId", convert("storeId", "string"))`
- ○ `transactionsDf.withColumn("storeId", col("storeId", "string"))`
- ○ `transactionsDf.withColumn("storeId", col("storeId").convert("string"))`
- ◉ `transactionsDf.withColumn("storeId", col("storeId").cast("string"))` **(Correct)**
- ○ `transactionsDf.withColumn("storeId", convert("storeId").as("string"))`

Question 34: **Correct**

Which of the following code blocks applies the boolean-returning Python function `evaluateTestSuccess` to column `storeId` of DataFrame `transactionsDf` as a user-defined function?

◉
```
1 from pyspark.sql import types as T
2 evaluateTestSuccessUDF = udf(evaluateTestSuccess,
T.BooleanType())
3 transactionsDf.withColumn("result",
evaluateTestSuccessUDF(col("storeId")))
```
(Correct)

○
```
1 evaluateTestSuccessUDF = udf(evaluateTestSuccess)
2 transactionsDf.withColumn("result", evaluateTestSuccessUDF(storeId))
```

○
```
1 from pyspark.sql import types as T
2 evaluateTestSuccessUDF = udf(evaluateTestSuccess, T.IntegerType())
3 transactionsDf.withColumn("result",
evaluateTestSuccess(col("storeId")))
```

○
```
1 evaluateTestSuccessUDF = udf(evaluateTestSuccess)
2 transactionsDf.withColumn("result",
evaluateTestSuccessUDF(col("storeId")))
```

○
```
1 from pyspark.sql import types as T
2 evaluateTestSuccessUDF = udf(evaluateTestSuccess, T.BooleanType())
3 transactionsDf.withColumn("result",
evaluateTestSuccess(col("storeId")))
```

Question 35: **Correct**

Which of the following code blocks returns a copy of DataFrame `transactionsDf` in which column `productId` has been renamed to `productNumber`?

- (●) `transactionsDf.withColumnRenamed("productId", "productNumber")` **(Correct)**
- ( ) `transactionsDf.withColumn("productId", "productNumber")`
- ( ) `transactionsDf.withColumnRenamed("productNumber", "productId")`
- ( ) `transactionsDf.withColumnRenamed(col(productId), col(productNumber))`
- ( ) `transactionsDf.withColumnRenamed(productId, productNumber)`

Question 36: **Correct**

Which of the following code blocks returns a copy of DataFrame `transactionsDf` that only includes columns `transactionId`, `storeId`, `productId` and `f`?

Sample of DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|   f|
+-------------+---------+-----+-------+---------+----+
|            1|        3|    4|     25|        1|null|
|            2|        6|    7|      2|        2|null|
|            3|        3| null|     25|        3|null|
+-------------+---------+-----+-------+---------+----+
```

- ( ) `transactionsDf.drop(col("value"), col("predError"))`
- (●) `transactionsDf.drop("predError", "value")` **(Correct)**
- ( ) `transactionsDf.drop(value, predError)`
- ( ) `transactionsDf.drop(["predError", "value"])`
- ( ) `transactionsDf.drop([col("predError"), col("value")])`

Question 37: **Correct**

The code block shown below should return a one-column DataFrame where the column `storeId` is converted to string type. Choose the answer that correctly fills the blanks in the code block to accomplish this.

`transactionsDf.__1__(__2__.__3__(__4__))`

- ○ 1. `select` 2. `col("storeId")` 3. `cast` 4. `StringType`
- ○ 1. `select` 2. `col("storeId")` 3. `as` 4. `StringType`
- ○ 1. `cast` 2. `"storeId"` 3. `as` 4. `StringType()`
- ◉ 1. `select` 2. `col("storeId")` 3. `cast` 4. `StringType()` **(Correct)**
- ○ 1. `select` 2. `storeId` 3. `cast` 4. `StringType()`

Question 38: **Correct**

Which of the following code blocks creates a new one-column, two-row DataFrame `dfDates` with column `date` of type `timestamp`?

○
```
dfDates = spark.createDataFrame(["23/01/2022 11:28:12","24/01/2022 10:58:34"], ["date"])
dfDates = dfDates.withColumn("date", to_timestamp("dd/MM/yyyy HH:mm:ss", "date"))
```

○
```
dfDates = spark.createDataFrame([("23/01/2022 11:28:12",),("24/01/2022 10:58:34",)], ["date"])
dfDates = dfDates.withColumnRenamed("date", to_timestamp("date", "yyyy-MM-dd HH:mm:ss"))
```

◉
```
dfDates = spark.createDataFrame([("23/01/2022 11:28:12",),("24/01/2022 10:58:34",)], ["date"])
dfDates = dfDates.withColumn("date", to_timestamp("date", "dd/MM/yyyy HH:mm:ss"))
```
(Correct)

○
```
dfDates = spark.createDataFrame(["23/01/2022 11:28:12","24/01/2022 10:58:34"], ["date"])
dfDates = dfDates.withColumnRenamed("date", to_datetime("date", "yyyy-MM-dd HH:mm:ss"))
```

○
```
dfDates = spark.createDataFrame([("23/01/2022 11:28:12",),("24/01/2022 10:58:34",)], ["date"])
```

Question 39: **Correct**

The code block displayed below contains an error. The code block should save DataFrame `transactionsDf` at path `path` as a parquet file, appending to any existing parquet file. Find the error.

Code block:

`transactionsDf.format("parquet").option("mode", "append").save(path)`

◉ **The code block is missing a reference to the `DataFrameWriter`.** (Correct)

○ **`save()` is evaluated lazily and needs to be followed by an action.**

○ **The mode option should be omitted so that the command uses the default mode.**

○ **The code block is missing a `bucketBy` command that takes care of partitions.**

○ **Given that the DataFrame should be saved as parquet file, `path` is being passed to the wrong method.**

Question 40: **Correct**

Which of the following code blocks returns a single row from DataFrame `transactionsDf`?

Full DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|   f|
+-------------+---------+-----+-------+---------+----+
|            1|        3|    4|     25|        1|null|
|            2|        6|    7|      2|        2|null|
|            3|        3| null|     25|        3|null|
|            4|     null| null|      3|        2|null|
|            5|     null| null|   null|        2|null|
|            6|        3|    2|     25|        2|null|
+-------------+---------+-----+-------+---------+----+
```

- ○ `transactionsDf.where(col("storeId").between(3,25))`
- ○ `transactionsDf.filter((col("storeId")!=25) | (col("productId")==2))`
- ◉ `transactionsDf.filter(col("storeId")==25).select("predError","storeId").distinct()` (Correct)
- ○ `transactionsDf.select("productId", "storeId").where("storeId == 2 OR storeId != 25")`
- ○ `transactionsDf.where(col("value").isNull()).select("productId", "storeId").distinct()`

Question 41: **Correct**

Which of the following code blocks returns approximately 1000 rows, some of them potentially being duplicates, from the 2000-row DataFrame `transactionsDf` that only has unique rows?

- (●) `transactionsDf.sample(True, 0.5)` **(Correct)**
- ( ) `transactionsDf.take(1000).distinct()`
- ( ) `transactionsDf.sample(False, 0.5)`
- ( ) `transactionsDf.take(1000)`
- ( ) `transactionsDf.sample(True, 0.5, force=True)`

Question 42: **Correct**

Which of the following code blocks returns a DataFrame showing the mean value of column "value" of DataFrame `transactionsDf`, grouped by its column `storeId`?

- ( ) `transactionsDf.groupBy(col(storeId).avg())`
- ( ) `transactionsDf.groupBy("storeId").avg(col("value"))`
- (●) `transactionsDf.groupBy("storeId").agg(avg("value"))` **(Correct)**
- ( ) `transactionsDf.groupBy("storeId").agg(average("value"))`
- ( ) `transactionsDf.groupBy("value").average()`

Question 43: **Correct**

Which of the following code blocks concatenates rows of DataFrames `transactionsDf` and `transactionsNewDf`, omitting any duplicates?

- ○ `transactionsDf.concat(transactionsNewDf).unique()`
- ◉ `transactionsDf.union(transactionsNewDf).distinct()` **(Correct)**
- ○ `spark.union(transactionsDf, transactionsNewDf).distinct()`
- ○ `transactionsDf.join(transactionsNewDf, how="union").distinct()`
- ○ `transactionsDf.union(transactionsNewDf).unique()`

Question 44: **Correct**

In which order should the code blocks shown below be run in order to assign `articlesDf` a DataFrame that lists all items in column `attributes` ordered by the number of times these items occur, from most to least often?

Sample of DataFrame `articlesDf`:

```
+------+-----------------------------+------------------+
|itemId|attributes                   |supplier          |
+------+-----------------------------+------------------+
|1     |[blue, winter, cozy]         |Sports Company Inc.|
|2     |[red, summer, fresh, cooling]|YetiX             |
|3     |[green, summer, travel]      |Sports Company Inc.|
+------+-----------------------------+------------------+
```

1. `articlesDf = articlesDf.groupby("col")`
2. `articlesDf = articlesDf.select(explode(col("attributes")))`
3. `articlesDf = articlesDf.orderBy("count").select("col")`
4. `articlesDf = articlesDf.sort("count",ascending=False).select("col")`
5. `articlesDf = articlesDf.groupby("col").count()`

- ○ **4, 5**
- ○ **2, 5, 3**
- ○ **5, 2**
- ○ **2, 3, 4**
- ◉ **2, 5, 4** (Correct)

Question 45: **Correct**

The code block shown below should set the number of partitions that Spark uses when shuffling data for joins or aggregations to 100. Choose the answer that correctly fills the blanks in the code block to accomplish this.

spark.sql.shuffle.partitions

```
__1__.__2__.__3__(__4__, 100)
```

- [x] 1. `spark`
  2. `conf`
  3. `set`
  4. `"spark.sql.shuffle.partitions"`

  (Correct)

- [ ] 1. `pyspark`
  2. `config`
  3. `set`
  4. `spark.shuffle.partitions`

- [ ] 1. `spark`
  2. `conf`
  3. `get`
  4. `"spark.sql.shuffle.partitions"`

- [ ] 1. `pyspark`
  2. `config`
  3. `set`
  4. `"spark.sql.shuffle.partitions"`

- [ ] 1. `spark`
  2. `conf`
  3. `set`
  4. `"spark.sql.aggregate.partitions"`

Question 46: **Correct**

The code block displayed below contains an error. The code block should read the csv file located at path `data/transactions.csv` into DataFrame `transactionsDf`, using the first row as column header and casting the columns in the most appropriate type. Find the error.

First 3 rows of `transactions.csv`:

```
transactionId;storeId;productId;name
1;23;12;green grass
2;35;31;yellow sun
3;23;12;green grass
```

Code block:

```
transactionsDf = spark.read.load("data/transactions.csv", sep=";", format="csv", header=True)
```

- ○ **The `DataFrameReader` is not accessed correctly.**
- ○ **The transaction is evaluated lazily, so no file will be read.**
- ○ **Spark is unable to understand the file type.**
- ○ **The code block is unable to capture all columns.**
- ◉ **The resulting DataFrame will not have the appropriate schema.** (Correct)

Question 47: **Correct**

The code block shown below should read all files with the file ending `.png` in directory `path` into Spark. Choose the answer that correctly fills the blanks in the code block to accomplish this.

```
spark.__1__.__2__(__3__).option(__4__, "*.png").__5__(path)
```

- ○ 1. `read()` 2. `format` 3. `"binaryFile"` 4. `"recursiveFileLookup"` 5. `load`
- ◉ 1. `read` 2. `format` 3. `"binaryFile"` 4. `"pathGlobFilter"` 5. `load` **(Correct)**
- ○ 1. `read` 2. `format` 3. `binaryFile` 4. `pathGlobFilter` 5. `load`
- ○ 1. `open` 2. `format` 3. `"image"` 4. `"fileType"` 5. `open`
- ○ 1. `open` 2. `as` 3. `"binaryFile"` 4. `"pathGlobFilter"` 5. `load`

Question 48: **Incorrect**

The code block displayed below contains an error. The code block should configure Spark so that DataFrames up to a size of 20 MB will be broadcast to all worker nodes when performing a join. Find the error.

Code block:

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 20)
```

- ◯ **Spark will only broadcast DataFrames that are much smaller than the default value.** (Correct)
- ◯ **The correct option to write configurations is through `spark.config` and not `spark.conf`.**
- ◉ **Spark will only apply the limit to threshold joins and not to other joins.** (Incorrect)
- ◯ **The passed limit has the wrong variable type.**
- ◯ **The command is evaluated lazily and needs to be followed by an action.**

Explanation
This is question is hard. Let's assess the different answers one-by-one.

Spark will only broadcast DataFrames that are much smaller than the default value.

This is correct. The default value is 10 MB (10485760 bytes). Since the configuration for `spark.sql.autoBroadcastJoinThreshold` expects a number in bytes (and not megabytes), the code block sets the limits to merely 20 bytes, instead of the requested 20 * 1024 * 1024 (= 20971520) bytes.

Question 49: **Correct**

The code block shown below should return a DataFrame with columns `transactionId`, `predError`, `value`, and `f` from DataFrame `transactionsDf`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

```
transactionsDf.__1__(__2__)
```

- ○ 1. `filter`
  2. `"transactionId", "predError", "value", "f"`

- ○ 1. `select`
  2. `"transactionId, predError, value, f"`

- ◉ 1. `select`
  2. `["transactionId", "predError", "value", "f"]`

  (Correct)

- ○ 1. `where`
  2. `col("transactionId"), col("predError"), col("value"), col("f")`

- ○ 1. `select`
  2. `col(["transactionId", "predError", "value", "f"])`

Question 50: **Incorrect**

Which of the following code blocks reads all CSV files in directory `filePath` into a single DataFrame, with column names defined in the CSV file headers?

Content of directory `filePath`:

```
1 _SUCCESS
2 _committed_2754546451699747124
3 _started_2754546451699747124
4 part-00000-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-298-1-c000.csv.gz
5 part-00001-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-299-1-c000.csv.gz
6 part-00002-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-300-1-c000.csv.gz
7 part-00003-tid-2754546451699747124-10eb85bf-8d91-4dd0-b60b-2f3c02eeecaa-301-1-c000.csv.gz
```

- ○ `spark.option("header",True).csv(filePath)`
- ◉ `spark.read.format("csv").option("header",True).option("compression","zip").load(filePath)` **(Incorrect)**
- ○ `spark.read().option("header",True).load(filePath)`
- ○ `spark.read.format("csv").option("header",True).load(filePath)` **(Correct)**
- ○ `spark.read.load(filePath)`

Explanation

The files in directory `filePath` are partitions of a DataFrame that have been exported using gzip compression. Spark automatically recognizes this situation and imports the CSV files as separate partitions into a single DataFrame. It is, however, necessary to specify that Spark should load the file headers in the CSV with the `header` option, which is set to `False` by default.

Question 51: **Incorrect**

The code block shown below should return a single-column DataFrame with a column named `consonant_ct` that, for each row, shows the number of consonants in column `itemName` of DataFrame `itemsDf`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

DataFrame `itemsDf`:

```
1 |  +------+----------------------------------+-----------------------------+-
  ------------------+
2 |  |itemId|itemName                          |attributes
  |supplier           |
3 |  +------+----------------------------------+-----------------------------+-
  ------------------+
4 |  |1     |Thick Coat for Walking in the Snow|[blue, winter, cozy]
  |Sports Company Inc.|
5 |  |2     |Elegant Outdoors Summer Dress     |[red, summer, fresh,
  cooling]|YetiX              |
6 |  |3     |Outdoors Backpack                 |[green, summer, travel]
  |Sports Company Inc.|
7 |  +------+----------------------------------+-----------------------------+-
  ------------------+
```

Code block:

```
itemsDf.select(__1__(__2__(__3__(__4__), "a|e|i|o|u|\s",
"")).__5__("consonant_ct"))
```

- ◯
  1. `length`
  2. `regexp_extract`
  3. `upper`
  4. `col("itemName")`
  5. `as`

- ◯
  1. `size`
  2. `regexp_replace`
  3. `lower`
  4. `"itemName"`
  5. `alias`

- ◯
  1. `lower`
  2. `regexp_replace`
  3. `length`
  4. `"itemName"`
  5. `alias`

- ◯
  1. `length`
  2. `regexp_replace`
  3. `lower` **(Correct)**
  4. `col("itemName")`
  5. `alias`

- ◉
  1. `size`
  2. `regexp_extract`
  3. `lower` **(Incorrect)**
  4. `col("itemName")`
  5. `alias`

Question 52: **Correct**

Which of the following code blocks produces the following output, given DataFrame `transactionsDf`?

Output:

```
root
 |-- transactionId: integer (nullable = true)
 |-- predError: integer (nullable = true)
 |-- value: integer (nullable = true)
 |-- storeId: integer (nullable = true)
 |-- productId: integer (nullable = true)
 |-- f: integer (nullable = true)
```

DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|   f|
+-------------+---------+-----+-------+---------+----+
|            1|        3|    4|     25|        1|null|
|            2|        6|    7|      2|        2|null|
|            3|        3| null|     25|        3|null|
+-------------+---------+-----+-------+---------+----+
```

- ◯ `transactionsDf.schema.print()`
- ◯ `transactionsDf.rdd.printSchema()`
- ◯ `transactionsDf.rdd.formatSchema()`
- ◉ `transactionsDf.printSchema()` (Correct)
- ◯ `print(transactionsDf.schema)`

Question 53: **Correct**

Which of the following code blocks returns a DataFrame that is an inner join of DataFrame `itemsDf` and DataFrame `transactionsDf`, on columns `itemId` and `productId`, respectively and in which every `itemId` just appears once?

- ```
  itemsDf.join(transactionsDf,
  "itemsDf.itemId==transactionsDf.productId").distinct("itemId")
  ```

- ```
  itemsDf.join(transactionsDf,
  itemsDf.itemId==transactionsDf.productId).dropDuplicates([
  "itemId"])
  ```
  (Correct)

- ```
  itemsDf.join(transactionsDf,
  itemsDf.itemId==transactionsDf.productId).dropDuplicates("itemId")
  ```

- ```
  itemsDf.join(transactionsDf,
  itemsDf.itemId==transactionsDf.productId,
  how="inner").distinct(["itemId"])
  ```

- ```
  itemsDf.join(transactionsDf,
  "itemsDf.itemId==transactionsDf.productId",
  how="inner").dropDuplicates(["itemId"])
  ```
  ```

Question 54: **Correct**

Which of the following code blocks reads in the parquet file stored at location `filePath`, given that all columns in the parquet file contain only whole numbers and are stored in the most appropriate format for this kind of data?

```
spark.read.schema(
    StructType(
        StructField("transactionId", IntegerType(), True),
        StructField("predError", IntegerType(), True)
    )).load(filePath)
```

```
spark.read.schema([
        StructField("transactionId", NumberType(), True),
        StructField("predError", IntegerType(), True)
    ]).load(filePath)
```

```
spark.read.schema(
    StructType([
        StructField("transactionId", StringType(), True),
        StructField("predError", IntegerType(), True)]
    )).parquet(filePath)
```

```
spark.read.schema(
    StructType([
        StructField("transactionId", IntegerType(), True),
        StructField("predError", IntegerType(), True)]
    )).format("parquet").load(filePath)
```

(Correct)

```
spark.read.schema([
        StructField("transactionId", IntegerType(), True),
        StructField("predError", IntegerType(), True)
    ]).load(filePath, format="parquet")
```

Question 55: **Correct**

Which of the following code blocks generally causes a great amount of network traffic?

`DataFrame.select()`

`DataFrame.coalesce()`

`DataFrame.collect()` **(Correct)**

`DataFrame.rdd.map()`

`DataFrame.count()`

Question 56: **Correct**

In which order should the code blocks shown below be run in order to return the number of records that are not empty in column `value` in the DataFrame resulting from an inner join of DataFrame `transactionsDf` and `itemsDf` on columns `productId` and `itemId`, respectively?

1. `.filter(~isnull(col('value')))`
2. `.count()`
3. `transactionsDf.join(itemsDf, col("transactionsDf.productId")==col("itemsDf.itemId"))`
4. `transactionsDf.join(itemsDf, transactionsDf.productId==itemsDf.itemId, how='inner')`
5. `.filter(col('value').isnotnull())`
6. `.sum(col('value'))`

- (●) **4, 1, 2** (Correct)
- ( ) **3, 1, 6**
- ( ) **3, 1, 2**
- ( ) **3, 5, 2**
- ( ) **4, 6**

Question 57: **Correct**

The code block displayed below contains an error. The code block should count the number of rows that have a `predError` of either `3` or `6`. Find the error.

Code block:

```
transactionsDf.filter(col('predError').in([3, 6])).count()
```

- ( ) **The number of rows cannot be determined with the `count()` operator.**
- ( ) **Instead of `filter`, the `select` method should be used.**
- (●) **The method used on column `predError` is incorrect.** (Correct)
- ( ) **Instead of a list, the values need to be passed as single arguments to the `in` operator.**
- ( ) **Numbers `3` and `6` need to be passed as string variables.**

Question 58: **Incorrect**

Which of the following code blocks returns a new DataFrame with the same columns as DataFrame `transactionsDf`, except for columns `predError` and `value` which should be removed?

- (●) `transactionsDf.drop(["predError", "value"])` **(Incorrect)**
- ( ) `transactionsDf.drop("predError", "value")` **(Correct)**
- ( ) `transactionsDf.drop(col("predError"), col("value"))`
- ( ) `transactionsDf.drop(predError, value)`
- ( ) `transactionsDf.drop("predError & value")`

Question 59: **Correct**

In which order should the code blocks shown below be run in order to read a JSON file from location `jsonPath` into a DataFrame and return only the rows that do not have value `3` in column `productId`?

1. `importedDf.createOrReplaceTempView("importedDf")`
2. `spark.sql("SELECT * FROM importedDf WHERE productId != 3")`
3. `spark.sql("FILTER * FROM importedDf WHERE productId != 3")`
4. `importedDf = spark.read.option("format", "json").path(jsonPath)`
5. `importedDf = spark.read.json(jsonPath)`

- ( ) **4, 1, 2**
- ( ) **5, 1, 3**
- ( ) **5, 2**
- ( ) **4, 1, 3**
- (●) **5, 1, 2** **(Correct)**

Question 60: **Incorrect**

The code block displayed below contains an error. The code block should return a DataFrame in which column `predErrorAdded` contains the results of Python function `add_2_if_geq_3` as applied to numeric and nullable column `predError` in DataFrame `transactionsDf`. Find the error.

Code block:

```
def add_2_if_geq_3(x):
  if x is None:
    return x
  elif x >= 3:
    return x+2
  return x

add_2_if_geq_3_udf = udf(add_2_if_geq_3)

transactionsDf.withColumnRenamed("predErrorAdded",
add_2_if_geq_3_udf(col("predError")))
```

- ○ **The operator used to adding the column does not add column `predErrorAdded` to the DataFrame.** (Correct)
- ○ **Instead of `col("predError")`, the actual DataFrame with the column needs to be passed, like so `transactionsDf.predError`.**
- ◉ **The `udf()` method does not declare a return type.** (Incorrect)
- ○ **UDFs are only available through the SQL API, but not in the Python API as shown in the code block.**
- ○ **The Python function is unable to handle `null` values, resulting in the code block crashing on execution.**

Explanation

Correct code block:

```
def add_2_if_geq_3(x):
  if x is None:
    return x
  elif x >= 3:
    return x+2
  return x

add_2_if_geq_3_udf = udf(add_2_if_geq_3)

transactionsDf.withColumn("predErrorAdded",
add_2_if_geq_3_udf(col("predError"))).show()
```

Instead of `withColumnRenamed`, you should use the `withColumn` operator.

Practice Test (Python) - 3 (Important)

Question 1: **Correct**

Which of the following describes a narrow transformation?

- ○ **A narrow transformation is an operation in which data is exchanged across partitions.**
- ○ **A narrow transformation is a process in which data from multiple RDDs is used.**
- ○ **A narrow transformation is a process in which 32-bit float variables are cast to smaller float variables, like 16-bit or 8-bit float variables.**
- ○ **A narrow transformation is an operation in which data is exchanged across the cluster.**
- ◉ **A narrow transformation is an operation in which no data is exchanged across the cluster.** (Correct)

Question 2: **Correct**

Which of the following statements about stages is correct?

- ○ **Different stages in a job may be executed in parallel.**
- ○ **Stages consist of one or more jobs.**
- ○ **Stages ephemerally store transactions, before they are committed through actions.**
- ◉ **Tasks in a stage may be executed by multiple machines at the same time.** (Correct)
- ○ **Stages may contain multiple actions, narrow, and wide transformations.**

Question 3: **Correct**

Which of the following describes tasks?

- ○ **A task is a command sent from the driver to the executors in response to a transformation.**
- ○ **Tasks transform jobs into DAGs.**
- ○ **A task is a collection of slots.**
- ○ **A task is a collection of rows.**
- ◉ **Tasks get assigned to the executors by the driver.** (Correct)

Question 4: **Correct**

Which of the following describes a difference between Spark's cluster and client execution modes?

- ○ **In cluster mode, the cluster manager resides on a worker node, while it resides on an edge node in client mode.**
- ○ **In cluster mode, executor processes run on worker nodes, while they run on gateway nodes in client mode.**
- ◉ **In cluster mode, the driver resides on a worker node, while it resides on an edge node in client mode.** (Correct)
- ○ **In cluster mode, a gateway machine hosts the driver, while it is co-located with the executor in client mode.**
- ○ **In cluster mode, the Spark driver is not co-located with the cluster manager, while it is co-located in client mode.**

Question 5: Incorrect

Which of the following describes Spark's standalone deployment mode?

- (●) **Standalone mode uses a single JVM to run Spark driver and executor processes.** (Incorrect)
- ( ) **Standalone mode means that the cluster does not contain the driver.**
- ( ) **Standalone mode is how Spark runs on YARN and Mesos clusters.**
- ( ) **Standalone mode uses only a single executor per worker per application.** (Correct)
- ( ) **Standalone mode is a viable solution for clusters that run multiple frameworks, not only Spark.**

Question 6: Incorrect

Which of the following describes properties of a shuffle?

- ( ) **Operations involving shuffles are never evaluated lazily.**
- ( ) **Shuffles involve only single partitions.**
- (●) **Shuffles belong to a class known as "full transformations".** (Incorrect)
- ( ) **A shuffle is one of many actions in Spark.**
- ( ) **In a shuffle, Spark writes data to disk.** (Correct)

Question 7: **Correct**

Which of the following statements about the differences between actions and transformations is correct?

- ○ **Actions are evaluated lazily, while transformations are not evaluated lazily.**
- ○ **Actions generate RDDs, while transformations do not.**
- ○ **Actions do not send results to the driver, while transformations do.**
- ○ **Actions can be queued for delayed execution, while transformations can only be processed immediately.**
- ◉ **Actions can trigger Adaptive Query Execution, while transformation cannot.** (Correct)

Question 8: **Incorrect**

Which of the following is a characteristic of the cluster manager?

- ○ **Each cluster manager works on a single partition of data.**
- ○ **The cluster manager receives input from the driver through the `SparkContext`.** (Correct)
- ◉ **The cluster manager does not exist in standalone mode.** (Incorrect)
- ○ **The cluster manager transforms jobs into DAGs.**
- ○ **In client mode, the cluster manager runs on the edge node.**

Question 9: **Incorrect**

Which of the following are valid execution modes?

- ○ **Kubernetes, Local, Client**
- ○ **Client, Cluster, Local** (Correct)
- ○ **Server, Standalone, Client**
- ○ **Cluster, Server, Local**
- ◉ **Standalone, Client, Cluster** (Incorrect)

Question 10: **Correct**

Which of the following describes Spark actions?

- ○ **Writing data to disk is the primary purpose of actions.**
- ○ **Actions are Spark's way of exchanging data between executors.**
- ◉ **The driver receives data upon request by actions.** (Correct)
- ○ **Stage boundaries are commonly established by actions.**
- ○ **Actions are Spark's way of modifying RDDs.**

Question 11: **Incorrect**

Which of the following statements about executors is correct?

- ○ **Executors are launched by the driver.**
- ○ **Executors stop upon application completion by default.** (Correct)
- ○ **Each node hosts a single executor.**
- ◉ **Executors store data in memory only.** (Incorrect)
- ○ **An executor can serve multiple applications.**

Question 12: **Correct**

Which of the following describes a valid concern about partitioning?

- ◉ **A shuffle operation returns 200 partitions if not explicitly set.** (Correct)
- ○ **Decreasing the number of partitions reduces the overall runtime of narrow transformations if there are more executors available than partitions.**
- ○ **No data is exchanged between executors when `coalesce()` is run.**
- ○ **Short partition processing times are indicative of low skew.**
- ○ **The `coalesce()` method should be used to increase the number of partitions.**

Question 13: **Incorrect**

Which of the following is characteristic of using accumulators?

- ◯ **Only unnamed accumulators can be inspected in the Spark UI.**
- ◉ **Only numeric values can be used in accumulators.** (Incorrect)
- ◯ **Accumulator values can only be read by the driver, but not by executors.** (Correct)
- ◯ **Accumulators do not obey lazy evaluation.**
- ◯ **Accumulators are difficult to use for debugging because they will only be updated once, independent if a task has to be re-run due to hardware failure.**

Question 14: **Incorrect**

Which of the following statements about reducing out-of-memory errors is incorrect?

- ◯ **Concatenating multiple string columns into a single column may guard against out-of-memory errors.** (Correct)
- ◯ **Reducing partition size can help against out-of-memory errors.**
- ◯ **Limiting the amount of data being automatically broadcast in joins can help against out-of-memory errors.**
- ◯ **Setting a limit on the maximum size of serialized data returned to the driver may help prevent out-of-memory errors.**
- ◉ **Decreasing the number of cores available to each executor can help against out-of-memory errors.** (Incorrect)

Question 15: **Incorrect**

Which of the following statements about storage levels is incorrect?

- ◯ **The `cache` operator on DataFrames is evaluated like a transformation.**
- ◉ **In client mode, DataFrames cached with the `MEMORY_ONLY_2` level will not be stored in the edge node's memory.** (Incorrect)
- ◯ **Caching can be undone using the `DataFrame.unpersist()` operator.**
- ◯ **`MEMORY_AND_DISK` replicates cached DataFrames both on memory and disk.** (Correct)
- ◯ **`DISK_ONLY` will not use the worker node's memory.**

Question 16: **Incorrect**

Which of the following is not a feature of Adaptive Query Execution?

- (●) **Replace a sort merge join with a broadcast join, where appropriate.** **(Incorrect)**
- ( ) **Coalesce partitions to accelerate data processing.**
- ( ) **Split skewed partitions into smaller partitions to avoid differences in partition processing time.**
- ( ) **Reroute a query in case of an executor failure.** **(Correct)**
- ( ) **Collect runtime statistics during query execution.**

Explanation

Reroute a query in case of an executor failure.

Correct. Although this feature exists in Spark, it is not a feature of Adaptive Query Execution. The cluster manager keeps track of executors and will work together with the driver to launch an executor and assign the workload of the failed executor to it (see also link below).

Replace a sort merge join with a broadcast join, where appropriate.

No, this is a feature of Adaptive Query Execution.

Coalesce partitions to accelerate data processing.

Wrong, Adaptive Query Execution does this.

Collect runtime statistics during query execution.

Incorrect, Adaptive Query Execution (AQE) collects these statistics to adjust query plans. This feedback loop is an essential part of accelerating queries via AQE.

Split skewed partitions into smaller partitions to avoid differences in partition processing time.

No, this is indeed a feature of Adaptive Query Execution. Find more information in the Databricks blog post linked below.

Question 17: **Incorrect**

The code block shown below should return a two-column DataFrame with columns `transactionId` and `supplier`, with combined information from DataFrames `itemsDf` and `transactionsDf`. The code block should merge rows in which column `productId` of DataFrame `transactionsDf` matches the value of column `itemId` in DataFrame `itemsDf`, but only where column `storeId` of DataFrame `transactionsDf` does not match column `itemId` of DataFrame `itemsDf`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Code block:

```
transactionsDf.__1__(itemsDf, __2__).__3__(__4__)
```

○
1. `join`
2. `transactionsDf.productId==itemsDf.itemId, how="inner"`
3. `select`
4. `"transactionId", "supplier"`

◉
1. `select`
2. `"transactionId", "supplier"`
3. `join`
4. `[transactionsDf.storeId!=itemsDf.itemId, transactionsDf.productId==itemsDf.itemId]`

(Incorrect)

○
1. `join`
2. `[transactionsDf.productId==itemsDf.itemId, transactionsDf.storeId!=itemsDf.itemId]`
3. `select`
4. `"transactionId", "supplier"`

(Correct)

○
1. `filter`
2. `"transactionId", "supplier"`
3. `join`
4. `"transactionsDf.storeId!=itemsDf.itemId, transactionsDf.productId==itemsDf.itemId"`

○
1. `join`
2. `transactionsDf.productId==itemsDf.itemId, transactionsDf.storeId!=itemsDf.itemId`
3. `filter`
4. `"transactionId", "supplier"`

Question 18: **Correct**

The code block shown below should return an exact copy of DataFrame `transactionsDf` that does not include rows in which values in column `storeId` have the value `25`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

- ○ `transactionsDf.remove(transactionsDf.storeId==25)`
- ◉ `transactionsDf.where(transactionsDf.storeId!=25)` **(Correct)**
- ○ `transactionsDf.filter(transactionsDf.storeId==25)`
- ○ `transactionsDf.drop(transactionsDf.storeId==25)`
- ○ `transactionsDf.select(transactionsDf.storeId!=25)`

Question 19: **Correct**

The code block shown below should return a copy of DataFrame `transactionsDf` with an added column `cos`. This column should have the values in column `value` converted to degrees and having the cosine of those converted values taken, rounded to two decimals. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Code block:

```
transactionsDf.__1__(__2__, round(__3__(__4__(__5__)),2))
```

- ◯
  1. `withColumn`
  2. `col("cos")`
  3. `cos`
  4. `degrees`
  5. `transactionsDf.value`

- ◯
  1. `withColumnRenamed`
  2. `"cos"`
  3. `cos`
  4. `degrees`
  5. `"transactionsDf.value"`

- ◉
  1. `withColumn`
  2. `"cos"`
  3. `cos`
  4. `degrees`
  5. `transactionsDf.value`

  (Correct)

- ◯
  1. `withColumn`
  2. `col("cos")`
  3. `cos`
  4. `degrees`
  5. `col("value")`

- ◯
  1. `withColumn`
  2. `"cos"`
  3. `degrees`
  4. `cos`
  5. `col("value")`

Question 20: **Correct**

The code block shown below should return the number of columns in the CSV file stored at location `filePath`. From the CSV file, only lines should be read that do not start with a `#` character. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Code block:

```
__1__(__2__.__3__.csv(filePath, __4__).__5__)
```

- 1. `size`
  2. `spark`
  3. `read()`
  4. `escape='#'`
  5. `columns`

- 1. `DataFrame`
  2. `spark`
  3. `read()`
  4. `escape='#'`
  5. `shape[0]`

- 1. `len`
  2. `pyspark`
  3. `DataFrameReader`
  4. `comment='#'`
  5. `columns`

- 1. `size`
  2. `pyspark`
  3. `DataFrameReader`
  4. `comment='#'`
  5. `columns`

- 1. `len`
  2. `spark`
  3. `read` (Correct)
  4. `comment='#'`
  5. `columns`

Question 21: **Incorrect**

Which of the following code blocks reads in the JSON file stored at `filePath`, enforcing the schema expressed in JSON format in variable `json_schema`, shown in the code block below?

Code block:

```
json_schema = """
{"type": "struct",
  "fields": [
    {
        "name": "itemId",
        "type": "integer",
        "nullable": true,
        "metadata": {}
    },
    {
        "name": "supplier",
        "type": "string",
        "nullable": true,
        "metadata": {}
    }
  ]
}
"""
```

- (●) `spark.read.json(filePath, schema=json_schema)` **(Incorrect)**
- ( ) `spark.read.schema(json_schema).json(filePath)`
- ( )
```
schema = StructType.fromJson(json.loads(json_schema))
spark.read.json(filePath, schema=schema)
```
(Correct)
- ( ) `spark.read.json(filePath, schema=schema_of_json(json_schema))`
- ( ) `spark.read.json(filePath, schema=spark.read.json(json_schema))`

Question 22: **Correct**

Which of the following code blocks applies the Python function `to_limit` on column `predError` in table `transactionsDf`, returning a DataFrame with columns `transactionId` and `result`?

- (●) (Correct)
```
spark.udf.register("LIMIT_FCN", to_limit)
spark.sql("SELECT transactionId, LIMIT_FCN(predError) AS result FROM transactionsDf")
```
- ( )
```
spark.udf.register("LIMIT_FCN", to_limit)
spark.sql("SELECT transactionId, LIMIT_FCN(predError) FROM transactionsDf AS result")
```
- ( )
```
spark.udf.register("LIMIT_FCN", to_limit)
spark.sql("SELECT transactionId, to_limit(predError) AS result FROM transactionsDf")
```
- ( ) `spark.sql("SELECT transactionId, udf(to_limit(predError)) AS result FROM transactionsDf")`
- ( )
```
spark.udf.register(to_limit, "LIMIT_FCN")
spark.sql("SELECT transactionId, LIMIT_FCN(predError) AS result FROM transactionsDf")
```

Question 23: **Correct**

Which of the following code blocks returns a single-row DataFrame that only has a column `corr` which shows the Pearson correlation coefficient between columns `predError` and `value` in DataFrame `transactionsDf`?

- ( ) `transactionsDf.select(corr(["predError", "value"]).alias("corr")).first()`
- ( ) `transactionsDf.select(corr(col("predError"), col("value")).alias("corr")).first()`
- ( ) `transactionsDf.select(corr(predError, value).alias("corr"))`
- (●) (Correct) `transactionsDf.select(corr(col("predError"), col("value")).alias("corr"))`
- ( ) `transactionsDf.select(corr("predError", "value"))`

Question 24: **Correct**

The code block shown below should return a DataFrame with all columns of DataFrame `transactionsDf`, but only maximum 2 rows in which column `productId` has at least the value `2`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

`transactionsDf.__1__(__2__).__3__`

- ○ 1. `where` 2. `"productId" > 2` 3. `max(2)`
- ○ 1. `where` 2. `transactionsDf[productId] >= 2` 3. `limit(2)`
- ○ 1. `filter` 2. `productId > 2` 3. `max(2)`
- ◉ 1. `filter` 2. `col("productId") >= 2` 3. `limit(2)` **(Correct)**
- ○ 1. `where` 2. `productId >= 2` 3. `limit(2)`

Question 25: **Incorrect**

Which of the following code blocks returns a single-column DataFrame of all entries in Python list `throughputRates` which contains only float-type values ?

- ○ `spark.createDataFrame((throughputRates), FloatType)`
- ○ `spark.createDataFrame(throughputRates, FloatType)`
- ○ `spark.DataFrame(throughputRates, FloatType)`
- ◉ `spark.createDataFrame(throughputRates)` **(Incorrect)**
- ○ `spark.createDataFrame(throughputRates, FloatType())` **(Correct)**

Question 26: **Correct**

Which of the following code blocks writes DataFrame `itemsDf` to disk at storage location `filePath`, making sure to substitute any existing data at that location?

- ◉ `itemsDf.write.mode("overwrite").parquet(filePath)` **(Correct)**
- ○ `itemsDf.write.option("parquet").mode("overwrite").path(filePath)`
- ○ `itemsDf.write(filePath, mode="overwrite")`
- ○ `itemsDf.write.mode("overwrite").path(filePath)`
- ○ `itemsDf.write().parquet(filePath, mode="overwrite")`

Question 27: **Correct**

The code block shown below should show information about the data type that column `storeId` of DataFrame `transactionsDf` contains. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Code block:

```
transactionsDf.__1__(__2__).__3__
```

- 1. `select`
  2. `"storeId"`
  3. `print_schema()`

- 1. `limit`
  2. `1`
  3. `columns`

- 1. `select`
  2. `"storeId"` (Correct)
  3. `printSchema()`

- 1. `limit`
  2. `"storeId"`
  3. `printSchema()`

- 1. `select`
  2. `storeId`
  3. `dtypes`

Question 28: **Correct**

The code block shown below should return a DataFrame with only columns from DataFrame `transactionsDf` for which there is a corresponding `transactionId` in DataFrame `itemsDf`. DataFrame `itemsDf` is very small and much smaller than DataFrame `transactionsDf`. The query should be executed in an optimized way. Choose the answer that correctly fills the blanks in the code block to accomplish this.

```
__1__.__2__(__3__, __4__, __5__)
```

- ◯
  1. `transactionsDf`
  2. `join`
  3. `broadcast(itemsDf)`
  4. `transactionsDf.transactionId==itemsDf.transactionId`
  5. `"outer"`

- ◯
  1. `transactionsDf`
  2. `join`
  3. `itemsDf`
  4. `transactionsDf.transactionId==itemsDf.transactionId`
  5. `"anti"`

- ◉
  1. `transactionsDf`
  2. `join`
  3. `broadcast(itemsDf)` **(Correct)**
  4. `"transactionId"`
  5. `"left_semi"`

- ◯
  1. `itemsDf`
  2. `broadcast`
  3. `transactionsDf`
  4. `"transactionId"`
  5. `"left_semi"`

- ◯
  1. `itemsDf`
  2. `join`
  3. `broadcast(transactionsDf)`
  4. `"transactionId"`
  5. `"left_semi"`

Question 29: **Incorrect**

The code block displayed below contains one or more errors. The code block should load parquet files at location `filePath` into a DataFrame, only loading those files that have been modified before 2029-03-20 05:44:46. Spark should enforce a schema according to the schema shown below. Find the error.

Schema:

```
root
 |-- itemId: integer (nullable = true)
 |-- attributes: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- supplier: string (nullable = true)
```

Code block:

```
schema = StructType([
  StructType("itemId", IntegerType(), True),
  StructType("attributes", ArrayType(StringType(), True), True),
  StructType("supplier", StringType(), True)
])

spark.read.options("modifiedBefore", "2029-03-
20T05:44:46").schema(schema).load(filePath)
```

- (●) **The `attributes` array is specified incorrectly and the syntax of the call to Spark's `DataFrameReader` is incorrect.** (Incorrect)
- ( ) **Columns in the schema definition use the wrong object type and the types defined for the columns do not match the schema.**
- ( ) **The data type of the `schema` is incompatible with the `schema()` operator and the modification date threshold is specified incorrectly.**
- ( ) **Columns in the schema definition use the wrong object type and the modification date threshold is specified incorrectly.** (Correct)
- ( ) **Columns in the schema are unable to handle empty values and the modification date threshold is specified incorrectly.**

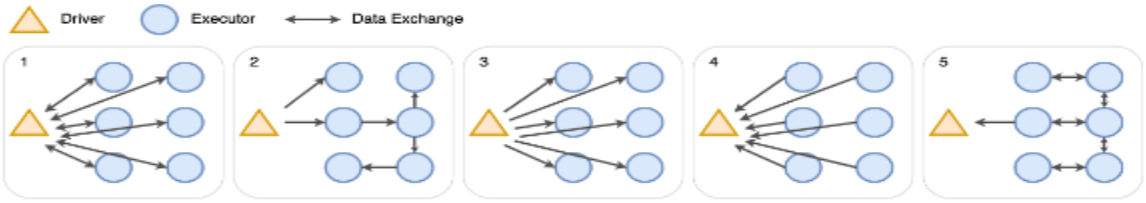
Question 30: **Correct**

Which of the following code blocks returns the number of unique values in column `storeId` of DataFrame `transactionsDf`?

- (●) `transactionsDf.select("storeId").dropDuplicates().count()` **(Correct)**
- ( ) `transactionsDf.select(count("storeId")).dropDuplicates()`
- ( ) `transactionsDf.select(distinct("storeId")).count()`
- ( ) `transactionsDf.dropDuplicates().agg(count("storeId"))`
- ( ) `transactionsDf.distinct().select("storeId").count()`

Question 31: **Incorrect**

Which of the elements in the labeled panels represent the operation performed for broadcast variables?



- ○ **2, 5**
- ◉ **3** **(Incorrect)**
- ○ **2, 3** **(Correct)**
- ○ **1, 2**
- ○ **1, 3, 4**

Explanation

2,3

Correct! Both panels 2 and 3 represent the operation performed for broadcast variables. While a broadcast operation may look like panel 3, with the driver being the bottleneck, it most probably looks like panel 2.

This is because the torrent protocol sits behind Spark's broadcast implementation. In the torrent protocol, each executor will try to fetch missing broadcast variables from the driver or other nodes, preventing the driver from being the bottleneck.

1,2

Wrong. While panel 2 may represent broadcasting, panel 1 shows bi-directional communication which does not occur in broadcast operations.

3

No. While broadcasting may materialize like shown in panel 3, its use of the torrent protocol also enables communciation as shown in panel 2 (see first explanation).

1,3,4

No. While panel 2 shows broadcasting, panel 1 shows bi-directional communication – not a characteristic of broadcasting. Panel 4 shows uni-directional communication, but in the wrong direction. Panel 4 resembles more an accumulator variable than a broadcast variable.

Question 32: **Correct**

Which of the following code blocks creates a new 6-column DataFrame by appending the rows of the 6-column DataFrame `yesterdayTransactionsDf` to the rows of the 6-column DataFrame `todayTransactionsDf`, ignoring that both DataFrames have different column names?

- ○ `union(todayTransactionsDf, yesterdayTransactionsDf)`
- ○ `todayTransactionsDf.unionByName(yesterdayTransactionsDf, allowMissingColumns=True)`
- ○ `todayTransactionsDf.unionByName(yesterdayTransactionsDf)`
- ○ `todayTransactionsDf.concat(yesterdayTransactionsDf)`
- ◉ `todayTransactionsDf.union(yesterdayTransactionsDf)` **(Correct)**

Question 33: **Correct**

Which of the following code blocks reads the parquet file stored at `filePath` into DataFrame `itemsDf`, using a valid schema for the sample of `itemsDf` shown below?

Sample of `itemsDf`:

```
+------+------------------------------+------------------+
|itemId|attributes                    |supplier          |
+------+------------------------------+------------------+
|1     |[blue, winter, cozy]          |Sports Company Inc.|
|2     |[red, summer, fresh, cooling]|YetiX              |
|3     |[green, summer, travel]       |Sports Company Inc.|
+------+------------------------------+------------------+
```

○
```
itemsDfSchema = StructType([
  StructField("itemId", IntegerType()),
  StructField("attributes", StringType()),
  StructField("supplier", StringType())])

itemsDf = spark.read.schema(itemsDfSchema).parquet(filePath)
```

○
```
itemsDfSchema = StructType([
  StructField("itemId", IntegerType),
  StructField("attributes", ArrayType(StringType)),
  StructField("supplier", StringType)])

itemsDf = spark.read.schema(itemsDfSchema).parquet(filePath)
```

○
```
itemsDf = spark.read.schema('itemId integer, attributes <string>, supplier string').parquet(filePath)
```

◉
```
itemsDfSchema = StructType([
  StructField("itemId", IntegerType()),
  StructField("attributes", ArrayType(StringType())),
  StructField("supplier", StringType())])

itemsDf = spark.read.schema(itemsDfSchema).parquet(filePath)
```
(Correct)

○
```
itemsDfSchema = StructType([
  StructField("itemId", IntegerType()),
  StructField("attributes", ArrayType([StringType()])),
  StructField("supplier", StringType())])

itemsDf = spark.read(schema=itemsDfSchema).parquet(filePath)
```

Question 34: **Incorrect**

The code block displayed below contains an error. The code block should trigger Spark to cache DataFrame `transactionsDf` in executor memory where available, writing to disk where insufficient executor memory is available, in a fault-tolerant way. Find the error.

Code block:

```
transactionsDf.persist(StorageLevel.MEMORY_AND_DISK)
```

- ◯ **Caching is not supported in Spark, data are always recomputed.**
- ◯ **Data caching capabilities can be accessed through the `spark` object, but not through the DataFrame API.**
- ◯ **The storage level is inappropriate for fault-tolerant storage.** (Correct)
- ◉ **The code block uses the wrong operator for caching.** (Incorrect)
- ◯ **The `DataFrameWriter` needs to be invoked.**

Explanation

The storage level is inappropriate for fault-tolerant storage.

Correct. Typically, when thinking about fault tolerance and storage levels, you would want to store redundant copies of the dataset. This can be achieved by using a storage level such as `StorageLevel.MEMORY_AND_DISK_2`.

The code block uses the wrong command for caching.

Wrong. In this case, `DataFrame.persist()` needs to be used, since this operator supports passing a storage level. `DataFrame.cache()` does not support passing a storage level.

Caching is not supported in Spark, data are always recomputed.

Incorrect. Caching is an important component of Spark, since it can help to accelerate Spark programs to great extent. Caching is often a good idea for datasets that need to be accessed repeatedly.

Data caching capabilities can be accessed through the `spark` object, but not through the `DataFrame` API.

No. Caching is either accessed through `DataFrame.cache()` or `DataFrame.persist()`.

Question 35: **Correct**

Which of the following code blocks returns all unique values across all values in columns `value` and `productId` in DataFrame `transactionsDf` in a one-column DataFrame?

- ○ `transactionsDf.select('value').join(transactionsDf.select('productId'), col('value')==col('productId'), 'outer')`
- ○ `transactionsDf.select(col('value'), col('productId')).agg({'*': 'count'})`
- ○ `transactionsDf.select('value', 'productId').distinct()`
- ◉ `transactionsDf.select('value').union(transactionsDf.select('productId')).distinct()` **(Correct)**
- ○ `transactionsDf.agg({'value': 'collect_set', 'productId': 'collect_set'})`

Question 36: **Correct**

Which of the following code blocks prints out in how many rows the expression `Inc.` appears in the string-type column `supplier` of DataFrame `itemsDf`?

○
```
counter = 0

for index, row in itemsDf.iterrows():
  if 'Inc.' in row['supplier']:
    counter = counter + 1

print(counter)
```

○
```
counter = 0

def count(x):
  if 'Inc.' in x['supplier']:
    counter = counter + 1

itemsDf.foreach(count)
print(counter)
```

○ `print(itemsDf.foreach(lambda x: 'Inc.' in x))`

○ `print(itemsDf.foreach(lambda x: 'Inc.' in x).sum())`

◉
```
accum=sc.accumulator(0)

def check_if_inc_in_supplier(row):
  if 'Inc.' in row['supplier']:
    accum.add(1)

itemsDf.foreach(check_if_inc_in_supplier)
print(accum.value)
```

(Correct)

Question 37: **Correct**

Which of the following code blocks shuffles DataFrame `transactionsDf`, which has 8 partitions, so that it has 10 partitions?

- ○ `transactionsDf.repartition(transactionsDf.getNumPartitions()+2)`
- ◉ `transactionsDf.repartition(transactionsDf.rdd.getNumPartitions()+2)` **(Correct)**
- ○ `transactionsDf.coalesce(10)`
- ○ `transactionsDf.coalesce(transactionsDf.getNumPartitions()+2)`
- ○ `transactionsDf.repartition(transactionsDf._partitions+2)`

Question 38: **Correct**

Which of the following code blocks returns a one-column DataFrame for which every row contains an array of all integer numbers from 0 up to but not including the number given in column `predError` of DataFrame `transactionsDf`, and `null` if `predError` is null?

Sample of DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|   f|
+-------------+---------+-----+-------+---------+----+
|            1|        3|    4|     25|        1|null|
|            2|        6|    7|      2|        2|null|
|            3|        3| null|     25|        3|null|
|            4|     null| null|      3|        2|null|
|            5|     null| null|   null|        2|null|
|            6|        3|    2|     25|        2|null|
+-------------+---------+-----+-------+---------+----+
```

◯

```
def count_to_target(target):
  if target is None:
    return

  result = [range(target)]
  return result

count_to_target_udf = udf(count_to_target, ArrayType[IntegerType])

transactionsDf.select(count_to_target_udf(col('predError')))
```

◯

```
def count_to_target(target):
  if target is None:
    return

  result = list(range(target))
  return result

transactionsDf.select(count_to_target(col('predError')))
```

◉

```
def count_to_target(target):
  if target is None:
    return

  result = list(range(target))
  return result

count_to_target_udf = udf(count_to_target,
ArrayType(IntegerType()))

transactionsDf.select(count_to_target_udf('predError'))
```

(Correct)

◯

```
def count_to_target(target):
  result = list(range(target))
  return result

count_to_target_udf = udf(count_to_target, ArrayType(IntegerType()))

df = transactionsDf.select(count_to_target_udf('predError'))
```

Question 39: **Incorrect**

Which of the following code blocks performs an inner join of DataFrames `transactionsDf` and `itemsDf` on columns `productId` and `itemId`, respectively, excluding columns `value` and `storeId` from DataFrame `transactionsDf` and column `attributes` from DataFrame `itemsDf`?

○ `transactionsDf.drop('value', 'storeId').join(itemsDf.select('attributes'), transactionsDf.productId==itemsDf.itemId)`

○
```
transactionsDf.createOrReplaceTempView('transactionsDf')
itemsDf.createOrReplaceTempView('itemsDf')

spark.sql("SELECT -value, -storeId FROM transactionsDf INNER JOIN itemsDf ON productId==itemId").drop("attributes")
```

◉ `transactionsDf.drop("value", "storeId").join(itemsDf.drop("attributes"), "transactionsDf.productId==itemsDf.itemId")` **(Incorrect)**

○
```
transactionsDf \
  .drop(col('value'), col('storeId')) \
  .join(itemsDf.drop(col('attributes')), col('productId')==col('itemId'))
```

○
```
transactionsDf.createOrReplaceTempView('transactionsDf')
itemsDf.createOrReplaceTempView('itemsDf')

statement = """
SELECT * FROM transactionsDf
INNER JOIN itemsDf
ON transactionsDf.productId==itemsDf.itemId
"""
spark.sql(statement).drop("value", "storeId", "attributes")
```
(Correct)

Question 40: **Correct**

The code block displayed below contains an error. The code block should configure Spark to split data in 20 parts when exchanging data between executors for joins or aggregations. Find the error.

Code block:

```
spark.conf.set(spark.sql.shuffle.partitions, 20)
```

- ◯ **The code block uses the wrong command for setting an option.**
- ◯ **The code block sets the wrong option.**
- ◉ **The code block expresses the option incorrectly.** (Correct)
- ◯ **The code block sets the incorrect number of parts.**
- ◯ **The code block is missing a parameter.**

Question 41: **Correct**

The code block displayed below contains an error. The code block should arrange the rows of DataFrame `transactionsDf` using information from two columns in an ordered fashion, arranging first by column `value`, showing smaller numbers at the top and greater numbers at the bottom, and then by column `predError`, for which all values should be arranged in the inverse way of the order of items in column `value`. Find the error.

Code block:

```
transactionsDf.orderBy('value', asc_nulls_first(col('predError')))
```

- ◯ **Two `orderBy` statements with calls to the individual columns should be chained, instead of having both columns in one `orderBy` statement.**
- ◯ **Column `value` should be wrapped by the `col()` operator.**
- ◉ **Column `predError` should be sorted in a descending way, putting nulls last.** (Correct)
- ◯ **Column `predError` should be sorted by `desc_nulls_first()` instead.**
- ◯ **Instead of `orderBy`, `sort` should be used.**

Question 42: **Correct**

The code block displayed below contains multiple errors. The code block should remove column `transactionDate` from DataFrame `transactionsDf` and add a column `transactionTimestamp` in which dates that are expressed as strings in column `transactionDate` of DataFrame `transactionsDf` are converted into unix timestamps. Find the errors.

Sample of DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+----------------+
|transactionId|predError|value|storeId|productId|   f| transactionDate|
+-------------+---------+-----+-------+---------+----+----------------+
|            1|        3|    4|     25|        1|null|2020-04-26 15:35|
|            2|        6|    7|      2|        2|null|2020-04-13 22:01|
|            3|        3| null|     25|        3|null|2020-04-02 10:53|
+-------------+---------+-----+-------+---------+----+----------------+
```

Code block:

```
transactionsDf = transactionsDf.drop("transactionDate")
transactionsDf["transactionTimestamp"] = unix_timestamp("transactionDate",
"yyyy-MM-dd")
```

- ◯ **Column `transactionDate` should be dropped after `transactionTimestamp` has been written. The string indicating the date format should be adjusted. The `withColumn` operator should be used instead of the existing column assignment. Operator `to_unixtime()` should be used instead of `unix_timestamp()`.**

- ◯ **Column `transactionDate` should be dropped after `transactionTimestamp` has been written. The `withColumn` operator should be used instead of the existing column assignment. Column `transactionDate` should be wrapped in a `col()` operator.**

- ◯ **Column `transactionDate` should be wrapped in a `col()` operator.**

- ◯ **The string indicating the date format should be adjusted. The `withColumnReplaced` operator should be used instead of the `drop` and assign pattern in the code block to replace column `transactionDate` with the new column `transactionTimestamp`.**

- ◉ **Column `transactionDate` should be dropped after `transactionTimestamp` has been written. The string indicating the date format should be adjusted. The `withColumn` operator should be used instead of the existing column assignment.** (Correct)

Question 43: **Correct**

Which of the following code blocks returns a new DataFrame in which column `attributes` of DataFrame `itemsDf` is renamed to `feature0` and column `supplier` to `feature1`?

- `itemsDf.withColumnRenamed(attributes, feature0).withColumnRenamed(supplier, feature1)`

- ```
  itemsDf.withColumnRenamed("attributes", "feature0")
  itemsDf.withColumnRenamed("supplier", "feature1")
  ```

- `itemsDf.withColumnRenamed(col("attributes"), col("feature0"), col("supplier"), col("feature1"))`

- `itemsDf.withColumnRenamed("attributes", "feature0").withColumnRenamed("supplier", "feature1")` **(Correct)**

- `itemsDf.withColumn("attributes", "feature0").withColumn("supplier", "feature1")`
  ```

Question 44: **Correct**

The code block displayed below contains multiple errors. The code block should return a DataFrame that contains only columns `transactionId`, `predError`, `value` and `storeId` of DataFrame `transactionsDf`. Find the errors.

Code block:

```
transactionsDf.select([col(productId), col(f)])
```

Sample of `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|   f|
+-------------+---------+-----+-------+---------+----+
|            1|        3|    4|     25|        1|null|
|            2|        6|    7|      2|        2|null|
|            3|        3| null|     25|        3|null|
+-------------+---------+-----+-------+---------+----+
```

- ◯ **The column names should be listed directly as arguments to the operator and not as a list.**
- ◉ **The `select` operator should be replaced by a `drop` operator, the column names should be listed directly as arguments to the operator and not as a list, and all column names should be expressed as strings without being wrapped in a `col()` operator.** (Correct)
- ◯ **The `select` operator should be replaced by a `drop` operator.**
- ◯ **The column names should be listed directly as arguments to the operator and not as a list and following the pattern of how column names are expressed in the code block, columns `productId` and `f` should be replaced by `transactionId`, `predError`, `value` and `storeId`.**
- ◯ **The `select` operator should be replaced by a `drop` operator, the column names should be listed directly as arguments to the operator and not as a list, and all `col()` operators should be removed.**

Question 45: **Correct**

Which of the following code blocks returns a DataFrame with approximately 1,000 rows from the 10,000-row DataFrame `itemsDf`, without any duplicates, returning the same rows even if the code block is run twice?

- ○ `itemsDf.sampleBy("row", fractions={0: 0.1}, seed=82371)`
- ◉ `itemsDf.sample(fraction=0.1, seed=87238)` **(Correct)**
- ○ `itemsDf.sample(fraction=1000, seed=98263)`
- ○ `itemsDf.sample(withReplacement=True, fraction=0.1, seed=23536)`
- ○ `itemsDf.sample(fraction=0.1)`

Question 46: **Incorrect**

Which of the following code blocks creates a new DataFrame with 3 columns, `productId`, `highest`, and `lowest`, that shows the biggest and smallest values of column `value` per value in column `productId` from DataFrame `transactionsDf`?

Sample of DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|   f|
+-------------+---------+-----+-------+---------+----+
|            1|        3|    4|     25|        1|null|
|            2|        6|    7|      2|        2|null|
|            3|        3| null|     25|        3|null|
|            4|     null| null|      3|        2|null|
|            5|     null| null|   null|        2|null|
|            6|        3|    2|     25|        2|null|
+-------------+---------+-----+-------+---------+----+
```

- ◯ `transactionsDf.max('value').min('value')`
- ◯ `transactionsDf.agg(max('value').alias('highest'), min('value').alias('lowest'))`
- ◯ `transactionsDf.groupby(col(productId)).agg(max(col(value)).alias("highest"), min(col(value)).alias("lowest"))`
- ◯ `transactionsDf.groupby('productId').agg(max('value').alias('highest'), min('value').alias('lowest'))` **(Correct)**
- ◉ `transactionsDf.groupby("productId").agg({"highest": max("value"), "lowest": min("value")})` **(Incorrect)**

Question 47: **Correct**

The code block displayed below contains an error. The code block should write DataFrame `transactionsDf` as a parquet file to location `filePath` after partitioning it on column `storeId`. Find the error.

Code block:

```
transactionsDf.write.partitionOn("storeId").parquet(filePath)
```

- ○ **The partitioning column as well as the file path should be passed to the `write()` method of DataFrame `transactionsDf` directly and not as appended commands as in the code block.**
- ○ **The `partitionOn` method should be called before the `write` method.**
- ○ **The operator should use the `mode()` option to configure the `DataFrameWriter` so that it replaces any existing files at location `filePath`.**
- ○ **Column `storeId` should be wrapped in a `col()` operator.**
- ◉ **No method `partitionOn()` exists for the `DataFrame` class, `partitionBy()` should be used instead.** (Correct)

Question 48: **Correct**

Which of the following code blocks reads in the JSON file stored at `filePath` as a DataFrame?

- (●) `spark.read.json(filePath)` **(Correct)**
- ( ) `spark.read.path(filePath, source="json")`
- ( ) `spark.read().path(filePath)`
- ( ) `spark.read().json(filePath)`
- ( ) `spark.read.path(filePath)`

Question 49: **Correct**

The code block shown below should add column `transactionDateForm` to DataFrame `transactionsDf`. The column should express the unix-format timestamps in column `transactionDate` as string type like `Apr 26 (Sunday)`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

```
transactionsDf.__1__(__2__, from_unixtime(__3__, __4__))
```

- ◯ 1. `withColumn` 2. `"transactionDateForm"` 3. `"MMM d (EEEE)"` 4. `"transactionDate"`

- ◯ 1. `select` 2. `"transactionDate"` 3. `"transactionDateForm"` 4. `"MMM d (EEEE)"`

- ◉ 1. `withColumn` 2. `"transactionDateForm"` 3. `"transactionDate"` 4. `"MMM d (EEEE)"` **(Correct)**

- ◯ 1. `withColumn` 2. `"transactionDateForm"` 3. `"transactionDate"` 4. `"MM d (EEE)"`

- ◯ 1. `withColumnRenamed` 2. `"transactionDate"` 3. `"transactionDateForm"` 4. `"MM d (EEE)"`

Explanation
Correct code block:

```
transactionsDf.withColumn("transactionDateForm",
from_unixtime("transactionDate", "MMM d (EEEE)"))
```

The question specifically asks about "adding" a column. In the context of all presented answers, `DataFrame.withColumn()` is the correct command for this. In theory, `DataFrame.select()` could also be used for this purpose, if all existing columns are selected and a new one is added. `DataFrame.withColumnRenamed()` is not the appropriate command, since it can only rename existing columns, but cannot add a new column or change the value of a column.

Once `DataFrame.withColumn()` is chosen, you can read in the documentation (see below) that the first input argument to the method should be the column name of the new column.

The final difficulty is the date format. The question indicates that the date format `Apr 26 (Sunday)` is desired. The answers give `"MMM d (EEEE)"` and `"MM d (EEE)"` as options. It can be hard to know the details of the date format that is used in Spark. Specifically, knowing the differences between `MMM` and `MM` is probably not something you deal with every day. But, there is an easy way to remember the difference: `M` (one letter) is usually the shortest form: `4` for April. `MM` includes padding: `04` for April. `MMM` (three letters) is the three-letter month abbreviation: `Apr` for April. And `MMMM` is the longest possible form: `April`. Knowing this four-letter sequence helps you select the correct option here.

Question 50: **Correct**

Which of the following code blocks shows the structure of a DataFrame in a tree-like way, containing both column names and types?

```
print(itemsDf.columns)
print(itemsDf.types)
```

- `itemsDf.printSchema()` **(Correct)**
- `spark.schema(itemsDf)`
- `itemsDf.rdd.printSchema()`
- `itemsDf.print.schema()`

Question 51: **Incorrect**

Which of the following code blocks reads in the two-partition parquet file stored at `filePath`, making sure all columns are included exactly once even though each partition has a different schema?

Schema of first partition:

```
root
 |-- transactionId: integer (nullable = true)
 |-- predError: integer (nullable = true)
 |-- value: integer (nullable = true)
 |-- storeId: integer (nullable = true)
 |-- productId: integer (nullable = true)
 |-- f: integer (nullable = true)
```

Schema of second partition:

```
root
 |-- transactionId: integer (nullable = true)
 |-- predError: integer (nullable = true)
 |-- value: integer (nullable = true)
 |-- storeId: integer (nullable = true)
 |-- rollId: integer (nullable = true)
 |-- f: integer (nullable = true)
 |-- tax_id: integer (nullable = false)
```

○ `spark.read.parquet(filePath, mergeSchema='y')`

○ `spark.read.option("mergeSchema", "true").parquet(filePath)` **(Correct)**

◉ `spark.read.parquet(filePath)` **(Incorrect)**

○
```
nx = 0
for file in dbutils.fs.ls(filePath):
  if not file.name.endswith(".parquet"):
    continue
  df_temp = spark.read.parquet(file.path)
  if nx == 0:
    df = df_temp
  else:
    df = df.union(df_temp)
  nx = nx+1
df
```

○
```
nx = 0
for file in dbutils.fs.ls(filePath):
  if not file.name.endswith(".parquet"):
    continue
  df_temp = spark.read.parquet(file.path)
  if nx == 0:
    df = df_temp
  else:
    df = df.join(df_temp, how="outer")
  nx = nx+1
df
```

Question 52: **Correct**

The code block shown below should add a column `itemNameBetweenSeparators` to DataFrame `itemsDf`. The column should contain arrays of maximum 4 strings. The arrays should be composed of the values in column `itemsDf` which are separated at `-` or whitespace characters. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Sample of DataFrame `itemsDf`:

```
+------+----------------------------------+-------------------+
|itemId|itemName                          |supplier           |
+------+----------------------------------+-------------------+
|1     |Thick Coat for Walking in the Snow|Sports Company Inc.|
|2     |Elegant Outdoors Summer Dress     |YetiX              |
|3     |Outdoors Backpack                 |Sports Company Inc.|
+------+----------------------------------+-------------------+
```

Code block:

```
itemsDf.__1__(__2__, __3__(__4__, "[\s\-]", __5__))
```

- (●) 1. `withColumn` 2. `"itemNameBetweenSeparators"` 3. `split` 4. `"itemName"` 5. `4` **(Correct)**
- ( ) 1. `withColumnRenamed` 2. `"itemNameBetweenSeparators"` 3. `split` 4. `"itemName"` 5. `4`
- ( ) 1. `withColumnRenamed` 2. `"itemName"` 3. `split` 4. `"itemNameBetweenSeparators"` 5. `4`
- ( ) 1. `withColumn` 2. `"itemNameBetweenSeparators"` 3. `split` 4. `"itemName"` 5. `5`

Question 53: **Incorrect**

The code block shown below should return a new 2-column DataFrame that shows one attribute from column `attributes` per row next to the associated `itemName`, for all suppliers in column `supplier` whose name includes `Sports`. Choose the answer that correctly fills the blanks in the code block to accomplish this.

Sample of DataFrame `itemsDf`:

```
+------+----------------------------------+----------------------------+-
------------------+
|itemId|itemName                          |attributes
|supplier           |
+------+----------------------------------+----------------------------+-
------------------+
|1     |Thick Coat for Walking in the Snow|[blue, winter, cozy]
|Sports Company Inc.|
|2     |Elegant Outdoors Summer Dress     |[red, summer, fresh,
cooling]|YetiX              |
|3     |Outdoors Backpack                 |[green, summer, travel]
|Sports Company Inc.|
+------+----------------------------------+----------------------------+-
------------------+
```

Code block:

```
itemsDf.__1__(__2__).select(__3__, __4__)
```

- (●) 1. `filter` 2. `col("supplier").isin("Sports")` 3. `"itemName"` 4. `explode(col("attributes"))` **(Incorrect)**
- ( ) 1. `where` 2. `col("supplier").contains("Sports")` 3. `"itemName"` 4. `"attributes"`
- ( ) 1. `where` 2. `col(supplier).contains("Sports")` 3. `explode(attributes)` 4. `itemName`
- ( ) 1. `where` 2. `"Sports".isin(col("Supplier"))` 3. `"itemName"` 4. `array_explode("attributes")`
- ( ) 1. `filter` 2. `col("supplier").contains("Sports")` 3. `"itemName"` 4. `explode("attributes")` **(Correct)**

Question 54: **Correct**

Which of the following code blocks immediately removes the previously cached DataFrame `transactionsDf` from memory and disk?

- ○ `array_remove(transactionsDf, "*")`
- ◉ `transactionsDf.unpersist()` **(Correct)**
- ○ `del transactionsDf`
- ○ `transactionsDf.clearCache()`
- ○ `transactionsDf.persist()`

Question 55: **Incorrect**

Which of the following code blocks returns a 2-column DataFrame that shows the distinct values in column `productId` and the number of rows with that `productId` in DataFrame `transactionsDf`?

- ○ `transactionsDf.count("productId").distinct()`
- ◉ `transactionsDf.groupBy("productId").agg(col("value").count())` **(Incorrect)**
- ○ `transactionsDf.count("productId")`
- ○ `transactionsDf.groupBy("productId").count()` **(Correct)**
- ○ `transactionsDf.groupBy("productId").select(count("value"))`

Question 56: **Correct**

Which of the following code blocks returns a DataFrame that matches the multi-column DataFrame `itemsDf`, except that integer column `itemId` has been converted into a string column?

- ○ `itemsDf.withColumn("itemId", convert("itemId", "string"))`
- ◉ `itemsDf.withColumn("itemId", col("itemId").cast("string"))` **(Correct)**
- ○ `itemsDf.select(cast("itemId", "string"))`
- ○ `itemsDf.withColumn("itemId", col("itemId").convert("string"))`
- ○ `spark.cast(itemsDf, "itemId", "string")`

Question 57: **Incorrect**

In which order should the code blocks shown below be run in order to create a DataFrame that shows the mean of column `predError` of DataFrame `transactionsDf` per column `storeId` and `productId`, where `productId` should be either `2` or `3` and the returned DataFrame should be sorted in ascending order by column `storeId`, leaving out any `nulls` in that column?

DataFrame `transactionsDf`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|   f|
+-------------+---------+-----+-------+---------+----+
|            1|        3|    4|     25|        1|null|
|            2|        6|    7|      2|        2|null|
|            3|        3| null|     25|        3|null|
|            4|     null| null|      3|        2|null|
|            5|     null| null|   null|        2|null|
|            6|        3|    2|     25|        2|null|
+-------------+---------+-----+-------+---------+----+
```

1. `.mean("predError")`
2. `.groupBy("storeId")`
3. `.orderBy("storeId")`
4. `transactionsDf.filter(transactionsDf.storeId.isNotNull())`
5. `.pivot("productId", [2, 3])`

- (◉) **4, 5, 2, 3, 1** (Incorrect)
- ( ) **4, 2, 1**
- ( ) **4, 1, 5, 2, 3**
- ( ) **4, 2, 5, 1, 3** (Correct)
- ( ) **4, 3, 2, 5, 1**

Explanation

Correct code block:

```
transactionsDf.filter(transactionsDf.storeId.isNotNull()).groupBy("storeId").pivot("productId", [2, 3]).mean("predError").orderBy("storeId")
```

Output of correct code block:

```
+-------+----+----+
|storeId|   2|   3|
+-------+----+----+
|      2| 6.0|null|
|      3|null|null|
|     25| 3.0| 3.0|
+-------+----+----+
```

This question is quite convoluted and requires you to think hard about the correct order of operations. The `pivot` method also makes an appearance - a method that you may not know all that much about (yet).

At the first position in all answers is code block 4, so the question is essentially just about the ordering of the remaining 4 code blocks.

The question states that the returned DataFrame should be sorted by column `storeId`. So, it should make sense to have code block 3 which includes the `orderBy` operator at the very end of the code block. This leaves you with only two answer options.

Now, it is useful to know more about the context of `pivot` in PySpark. A common pattern is `groupBy`, `pivot`, and then another aggregating function, like `mean`. In the documentation linked below you can see that `pivot` is a method of `pyspark.sql.GroupedData` - meaning that before pivoting, you have to use `groupBy`. The only answer option matching this requirement is the one in which code block 2 (which includes `groupBy`) is stated before code block 5 (which includes `pivot`).

Question 58: **Correct**

The code block displayed below contains an error. The code block should combine data from DataFrames `itemsDf` and `transactionsDf`, showing all rows of DataFrame `itemsDf` that have a matching value in column `itemId` with a value in column `transactionsId` of DataFrame `transactionsDf`. Find the error.

Code block:

```
itemsDf.join(itemsDf.itemId==transactionsDf.transactionId)
```

- (●) **The `join` statement is incomplete.** **(Correct)**
- ( ) **The `union` method should be used instead of `join`.**
- ( ) **The join method is inappropriate.**
- ( ) **The `merge` method should be used instead of `join`.**
- ( ) **The join expression is malformed.**

Question 59: **Correct**

The code block displayed below contains an error. The code block should merge the rows of DataFrames `transactionsDfMonday` and `transactionsDfTuesday` into a new DataFrame, matching column names and inserting `null` values where column names do not appear in both DataFrames. Find the error.

Sample of DataFrame `transactionsDfMonday`:

```
+-------------+---------+-----+-------+---------+----+
|transactionId|predError|value|storeId|productId|   f|
+-------------+---------+-----+-------+---------+----+
|            5|     null| null|   null|        2|null|
|            6|        3|    2|     25|        2|null|
+-------------+---------+-----+-------+---------+----+
```

Sample of DataFrame `transactionsDfTuesday`:

```
+-------+-------------+---------+-----+
|storeId|transactionId|productId|value|
+-------+-------------+---------+-----+
|     25|            1|        1|    4|
|      2|            2|        2|    7|
|      3|            4|        2| null|
|   null|            5|        2| null|
+-------+-------------+---------+-----+
```

Code block:

```
sc.union([transactionsDfMonday, transactionsDfTuesday])
```

- ○ **The DataFrames' RDDs need to be passed into the `sc.union` method instead of the DataFrame variable names.**
- ○ **Instead of `union`, the `concat` method should be used, making sure to not use its default arguments.**
- ○ **Instead of the Spark context, `transactionDfMonday` should be called with the `join` method instead of the `union` method, making sure to use its default arguments.**
- ○ **Instead of the Spark context, `transactionDfMonday` should be called with the `union` method.**
- ◉ **Instead of the Spark context, `transactionDfMonday` should be called with the `unionByName` method instead of the `union` method, making sure to not use its default arguments.** (Correct)

Question 60: **Incorrect**

Which of the following code blocks displays various aggregated statistics of all columns in DataFrame `transactionsDf`, including the standard deviation and minimum of values in each column?

- (●) `transactionsDf.summary()` **(Incorrect)**
- ( ) `transactionsDf.agg("count", "mean", "stddev", "25%", "50%", "75%", "min")`
- ( ) `transactionsDf.summary("count", "mean", "stddev", "25%", "50%", "75%", "max").show()`
- ( ) `transactionsDf.agg("count", "mean", "stddev", "25%", "50%", "75%", "min").show()`
- ( ) `transactionsDf.summary().show()` **(Correct)**

Explanation

The `DataFrame.summary()` command is very practical for quickly calculating statistics of a DataFrame. You need to call `.show()` to display the results of the calculation. By default, the command calculates various statistics (see documentation linked below), including standard deviation and minimum. Note that the answer that lists many options in the `summary()` parentheses does not include the minimum, which is asked for in the question.

Answer options that include `agg()` do not work here as shown, since `DataFrame.agg()` expects more complex, column-specific instructions on how to aggregate values.

TEST
Practice Test
Python -1

Question 1: **Correct**

If we want to create a constant integer 1 as a new column '**new_column**' in a dataframe `df`, which code block we should select ?

- ○ `df.withColumn("new_column", lit("1"))`
- ◉ `df.withColumn("new_column", lit(1))` (Correct)
- ○ `df.withColumn("new_column", 1)`
- ○ `df.withColumnRenamed('new_column', lit(1))`
- ○ `df.withColumn(new_column, lit(1))`

Question 2: **Correct**

Given the code block down below, a database test and a dataframe containing nulls, identify the error.

```
1  def strlen(s):
2    return len(s)
1  spark.udf.register("strlen", strlen)
2  spark.sql("select s from test where strlen(s) > 1")
```

- ○ **We need to use function 'query' instead of 'sql' to query table test.**
- ◉ **This WHERE clause does not guarantee the strlen UDF to be invoked after filtering out nulls. So we will have null pointer exception.** (Correct)
- ○ **There is no problem with this query.**

Question 3: **Correct**

The code block shown below intends to return a new DataFrame with column “old” renamed to “new” but it contains an error. Identify the error.

```
df.withColumnRenamed(“new”, “old”)
```

- ◯ **There should be no quotes for the column names.**
  `df.withColumnRenamed(new, old)`
- ◉ **Parameters are inverted; correct usage is**
  `df.withColumnRenamed(“old”, “new”)`
  (Correct)
- ◯ **We need to add ‘col’ to specifiy that it’s a column.**
  `df.withColumnRenamed(col(“new”), col(“old”))`
- ◯ **WithColumnRenamed is not a valid fonction , we need to use**
  `df.withColumnRenamed(“new”, “old”)`

Question 4: **Correct**

Which of the following describes the relationship between worker nodes and executors?

- ◯ **Executors and worker nodes are not related.**
- ◉ **An executor is a Java Virtual Machine (JVM) running on a worker node.** **(Correct)**
- ◯ **A worker node is a Java Virtual Machine (JVM) running on an executor.**
- ◯ **There are always the same number of executors and worker nodes.**
- ◯ **There are always more worker nodes than executors.**

Question 5: **Correct**

There is a global temp view named `'my_global_view'`. If I want to query this view within spark, which command I should choose ?

- ◯ **spark.read.view("my_global_view")**
- ◉ **spark.read.table("global_temp.my_global_view")** (Correct)
- ◯ **spark.read.view("global_temp.my_global_view")**
- ◯ **spark.read.table("my_global_view")**

Question 6: **Correct**

You have a need to sort a dataframe named **df** which has some null values on column **a**. You want the null values to appear first, and then the rest of the rows should be ordered descending based on the column a. Choose the right code block to achieve your goal.

- ◯ `df.sortBy(desc_nulls_first("a"))`
- ◯ `df.orderBy(desc_nulls_first(a))`
- ◯ **It is not possible to sort, when there are null values on the specified column.**
- ◉ `df.orderBy(df.a.desc_nulls_first())` (Correct)
- ◯ `df.orderBy(desc("a"))`

Question 7: **Correct**

Which of the following statement is true for broadcast variables ?

- (●) **Broadcast variables are shared, immutable variables that are cached on every machine in the cluster instead of serialized with every single task** (Correct)
- ( ) **It is a way of updating a value inside of a variety of transformations and propagating that value to the driver node in an efficient and fault-tolerant way.**
- ( ) **It provides a mutable variable that a Spark cluster can safely update on a per-row basis**
- ( ) **The canonical use case is to pass around a extermely large table that does not fit in memory on the executors.**

Question 8: **Correct**

Given an instance of SparkSession named spark, and the following DataFrame named

```
from pyspark.sql.functions import sort_array, collect_list
import pyspark.sql.functions as f

rawData = [   (1, 1000, "Apple", 0.76),   (2, 1000, "Apple", 0.11),   (1, 2000, "Orange", 0.98),   (1, 3000, "Banana", 0.24),   (2, 3000, "Banana", 0.99) ]

dfA = spark.createDataFrame(rawData).toDF("UserKey", "ItemKey", "ItemName", "Score")
```

Select the code fragment that produces the following result:

```
+-------+-----------------------------------------------------------------+
|UserKey|Collection                                                       |
+-------+-----------------------------------------------------------------+
|1      |[[0.98, 2000, Orange], [0.76, 1000, Apple], [0.24, 3000, Banana]]|
|2      |[[0.99, 3000, Banana], [0.11, 1000, Apple]]                      |
```

◉

```
dfA.groupBy("UserKey")
  .agg(sort_array(collect_list(f.struct("Score", "ItemKey", "ItemName")), False))
  .toDF("UserKey", "Collection")
  .show(20, False)
```

(Correct)

○

```
dfA.groupBy("UserKey")
  .agg(collect_list(struct("Score", "ItemKey", "ItemName")))
  .toDF("UserKey", "Collection")
  .show(20, False)
```

○

```
import org.apache.spark.sql.expressions.Window
dfA.withColumn("Collection",collect_list(struct("Score", "ItemKey", "ItemName")).over(Window.partitionBy("ItemKey")))
  .select("UserKey", "Collection")
  .show(20, False)
```

○

```
dfA.groupBy("UserKey", "ItemKey", "ItemName")
  .agg(sort_array(collect_list(struct("Score", "ItemKey", "ItemName")), false))
  .drop("ItemKey", "ItemName")       .toDF("UserKey", "Collection")

  .show(20, False)
```

Question 9: **Correct**

The code block shown below should return a DataFrame with column only aSquared dropped from DataFrame df. Choose the response that correctly fills in the numbered blanks within the code block to complete this task.

Code block:

```
df.__1__(__2__)
```

- ◯ **1. drop**
  2. aSquared
- ◯ **1. remove**
  2. aSquared
- ◉ **1. drop**
  2. “aSquared” **(Correct)**
- ◯ **1. remove**
  2. “aSquared”

Question 10: **Incorrect**

The code block shown below contains an error. The code block is intended to write a text file in the path. Identify the error.

```
1  df = spark.range(1 * 10000000).toDF("id").withColumn("s2", col("id") *
col("id")).withColumn("s3",lit(1))
2
3  df.write.text(“my_file.txt”)
```

- ◯ **For text files, we can only have one column in the dataframe that we want to write.** **(Correct)**
- ◯ **The maximum limit of lines in a text file has been reached and therefore we cannot create a text file.**
- ◉ **We need to use save instead of write function.** **(Incorrect)**
- ◯ **We need to provide at least one option.**

Question 11: **Incorrect**

What are the possible strategies in order to decrease garbage collection time ?

- [x] **Increase java heap space size** (Correct)
- [ ] **Persist objects in serialized form** (Correct)
- [x] **Create fewer objects** (Correct)

Explanation

JVM garbage collection can be a problem when you have large “churn” in terms of the RDDs stored by your program. When Java needs to evict old objects to make room for new ones, it will need to trace through all your Java objects and find the unused ones. The main point to remember here is that the cost of garbage collection is proportional to the number of Java objects, so using data structures with fewer objects (e.g. an array of Ints instead of a LinkedList) greatly lowers this cost.
https://spark.apache.org/docs/latest/tuning.html#garbage-collection-tuning

Question 12: **Correct**

What won't cause a full shuffle knowing that dataframe ‘df’ has 8 partitions ?

- ◯ **All of them will cause a full shuffle.**
- ◯ **df.repartition(12)**
- ◉ **df.coalesce(4)** (Correct)

Explanation

Coalse function avoids a full shuffle if it's known that the number is decreasing then the executor can safely keep data on the minimum number of partitions, only moving the data off the extra nodes, onto the nodes that we kept.

Question 13: **Correct**

Which of the following is true for driver ?

- ◯ **Is a chunk of data that sit on a single machine in a cluster.**
- ◉ **Responsible for assigning work that will be completed in parallel.** (Correct)
- ◯ **Responsible for allocating resources for worker nodes.**
- ◯ **Responsible for executing work that will be completed in parallel.**
- ◯ **Reports the state of some computation back to a central system.**

Question 14: **Correct**

Which of the following code blocks reads from a csv file where values are separated with **';'** ?

- ◯ **spark.load.format("csv").option("header", "true").option("inferSchema", "true").read(file)**
- ◯ **spark.read.format("csv").option("header", "true").option("inferSchema", "true").load(file)**
- ◉ **spark.read.format("csv").option("header", "true").option("inferSchema", "true").option("sep", ";").load(file)** (Correct)
- ◯ **spark.read.format("csv").option("header", "true").option("inferSchema", "true").option("sep", "true").toDf(file)**

Question 15: **Correct**

tableA is a DataFrame consisting of 20 fields and 40 billion rows of data with a surrogate key field. tableB is a DataFrame functioning as a lookup table for the surrogate key consisting of 2 fields and 5,000 rows. If the in-memory size of tableB is 22MB, what occurs when the following code is executed:?

```
df = tableA.join(tableB, "primary_key")
```

- ○ **An exception will be thrown due to tableB being greater than the 10MB default threshold for a broadcast join.**
- ○ **The contents of tableB will be partitioned so that each of the keys that need to be joined on in tableA partitions on each executor will match.**
- ◉ **A non-broadcast join will be executed with a shuffle phase since the broadcast table is greater than the 10MB default threshold and the broadcast hint was not specified.** (Correct)
- ○ **The contents of tableB will be replicated and sent to each executor to eliminate the need for a shuffle stage during the join.**

Question 16: **Incorrect**

The code blown down below intends to join df1 with df2 with inner join but it contains an error. Identify the error.

```
d1.join(d2, “inner”, d1.col(“id”) === df2.col(“id”))
```

- ○ **Syntax is not correct**
  ```
  d1.join(d2, d1.col(“id”) == df2.col(“id"), “inner”)
  ```
  (Correct)
- ○ **There should be two == instead of ===. So the correct query is**
  ```
  d1.join(d2, “inner”, d1.col(“id”) == df2.col(“id"))
  ```
- ○ **We cannot do inner join in spark 3.0, but it is in the roadmap.**
- ◉ **The join type is not in right order. The correct query should be**
  ```
  d2.join(d1, d1.col(“id”) === df2.col(“id"), “inner”)
  ```
  (Incorrect)

Question 17: **Incorrect**

Choose the right order of commands in order to query table '**test**' in database '**db**'

```
1. Use db
2. Switch db
3. Select db
4. Select * from test
5. Select * from db
```

- (●) **1, 5** **(Incorrect)**
- ( ) **3, 5**
- ( ) **3, 4**
- ( ) **2, 4**
- ( ) **2, 5**
- ( ) **1, 4** **(Correct)**

Question 18: **Correct**

Which of the following 3 DataFrame operations are classified as a wide transformation ? Choose 3 answers:

- [x] **orderBy()** **(Correct)**
- [x] **repartition()** **(Correct)**
- [ ] **cache()**
- [ ] **filter()**
- [x] **distinct()** **(Correct)**
- [ ] **drop()**

Question 18: Correct

Which of the following 3 DataFrame operations are classified as a wide transformation ? Choose 3 answers:

- [x] **orderBy()** (Correct)
- [x] **repartition()** (Correct)
- [ ] **cache()**
- [ ] **filter()**
- [x] **distinct()** (Correct)
- [ ] **drop()**

Question 19: Incorrect

Which of the following statements about Spark accumulator variables is NOT true?

- ( ) **You can define your own custom accumulator class by extending org.apache.spark.util.AccumulatorV2 in Java or Scala or pyspark.AccumulatorParam in Python.**
- ( ) **The Spark UI displays all accumulators used by your application.** (Correct)
- (•) **For accumulator updates performed inside actions only, Spark guarantees that each task's update to the accumulator will be applied only once, meaning that restarted tasks will not update the value.** (Incorrect)
- ( ) **In transformations, each task's update can be applied more than once if tasks or job stages are re-executed.**
- ( ) **Accumulators provide a shared, mutable variable that a Spark cluster can safely update on a per-row basis.**

Question 20: **Correct**

Which property is used to scale up and down dynamically based on applications' current number of pending tasks in a spark cluster ?

- ◯ **There is no need to set a property since spark is by default capable of resizing**
- ◉ **Dynamic allocation** (Correct)
- ◯ **Fair Scheduler**

Explanation

If you would like to run multiple Spark Applications on the same cluster, Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload. This means that your application can give resources back to the cluster if they are no longer used, and request them again later when there is demand. This feature is particularly useful if multiple applications share resources in your Spark cluster. This feature is disabled by default and available on all coarse-grained cluster managers; that is, standalone mode, YARN mode, and Mesos coarse-grained mode. There are two requirements for using this feature. First, your application must set spark.dynamicAllocation.enabled to true. Second, you must set up an external shuffle service on each worker node in the same cluster and set spark.shuffle.service.enabled to true in your application. The purpose of the external shuffle service is to allow executors to be removed without deleting shuffle files written by them. The Spark Fair Scheduler specifies resource pools and allocates jobs to different resource pools to achieve resource scheduling within an application. In this way, the computing resources are effectively used and the runtime of jobs is balanced, ensuring that the subsequently-submitted jobs are not affected by over-loaded jobs.

Question 21: **Incorrect**

Given that the number of partitions of dataframe df is 4 and we want to write a parquet file in a given path. Choose the correct number of files after a successful write operation.

- ◯ **8**
- ◯ **4** (Correct)
- ◉ **1** (Incorrect)

Explanation

We control the parallelism of files that we write by controlling the partitions prior to writing and therefore the number of partitions before writing equals to number of files created after the write operation.

Question 22: **Correct**

Which of the following describes a 'job' in Spark best ?

- ◯ **A unit of work that will be sent to one executor.**
- ◉ **A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect).** (Correct)
- ◯ **A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them.**
- ◯ **User program built on Spark. Consists of a driver program and executors on the cluster.**
- ◯ **An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)**

Question 23: **Incorrect**

Which of the following describe optimizations enabled by adaptive query execution (AQE)? Choose two.

- [ ] **AQE allows you to dynamically convert physical plans to RDDs**
- [x] **AQE allows you to dynamically reorganize query orders.** **(Incorrect)**
- [ ] **AQE allows you to dynamically coalesce shuffle partitions** **(Correct)**
- [x] **AQE allows you to dynamically switch join strategies.** **(Correct)**
- [ ] **AQE allows you to dynamically select physical plans based on cost.**

Explanation

AQE attempts to to do the following at runtime:

1. Reduce the number of reducers in the shuffle stage by decreasing the number of shuffle partitions.

2. Optimize the physical execution plan of the query, for example by converting a `SortMergeJoin` into a `BroadcastHashJoin` where appropriate.

3. Handle data skew during a join.

Hence the following responses are correct;

1. AQE allows you to dynamically switch join strategies.

2. AQE allows you to dynamically coalesce shuffle partitions.

Spark catalyst optimizer let's you do;

1. Dynamically convert physical plans to RDDs.

2.Dynamically reorganize query orders.

3. Dynamically select physical plans based on cost.

Question 24: **Correct**

Which of the following code blocks returns a DataFrame with a new column aSquared and all previously existing columns from DataFrame **df** given that df has a column named **a** ?

- ○ `df.withColumn("aSquared", col(a) * col(a))`
- ◉ `df.withColumn("aSquared", col("a") * col("a"))` **(Correct)**
- ○ `df.withColumn(aSquared, col("a") * col("a"))`
- ○ `df.withColumn(col("a") * col("a"), "aSquared")`
- ○ `df.withColumn(aSquared, col(a) * col(a))`

Question 25: **Correct**

The following statement will create a managed table

```
dataframe.write.option('path',
"/my_paths/").saveAsTable("managed_my_table")
```

- ○ **TRUE**
- ◉ **FALSE** **(Correct)**

Explanation

Spark manages the metadata, while you control the data location. As soon as you add 'path' option in dataframe writer it will be treated as global external/unmanaged table. When you drop table only metadata gets dropped. A global unmanaged/external table is available across all clusters.

Question 26: **Correct**

Which of the following are correct for slots ?

- [x] **Spark parallelizes via slots.** (Correct)
- [x] **Each executor has a number of slots.** (Correct)
- [ ] **All of the answers are correct.**
- [x] **Each slot can be assigned a task.** (Correct)
- [ ] **It is interchangeable with tasks.**

Explanation

Slots are not the same thing as executors. Executors could have multiple slots in them, and tasks are executed on slots. Review well this concept for the exam.
https://spark.apache.org/docs/latest/cluster-overview.html

Question 27: **Correct**

Which of the following DataFrame operation is classified as a narrow transformation ?

- ( ) **orderBy()**
- ( ) **coalse()**
- (•) **filter()** (Correct)
- ( ) **distinct()**
- ( ) **repartition()**

Question 28: **Incorrect**

Consider the following DataFrame:

```
import org.apache.spark.sql.functions._

data = [   ("Ali", 0, [100]),   ("Barbara", 1, [300, 250, 100]),
("Cesar", 1, [350, 100]),   ("Dongmei", 1, [400, 100]),   ("Eli", 2, [250]),
   ("Florita", 2, [500, 300, 100]),   ("Gatimu", 3, [300, 100]) ]

peopleDF = spark.createDataFrame(data).toDF("name", "department", "score")
```

Select the code fragment that produces the following result:

```
+----------+-------+-------+
|department| name|highest|
+----------+-------+-------+
| 0| Ali| 100|
| 1|Dongmei| 400|
| 2|Florita| 500|
| 3| Gatimu| 300|
+----------+-------+-------+
```

○

```
from pyspark.sql import Window
from pyspark.sql.functions import *

windowSpec =
Window.partitionBy("department").orderBy(col("score").desc())

peopleDF.withColumn("score", explode(col("score")))
.select(col("department"),col("name"),dense_rank()
.over(windowSpec).alias("rank"),max(col("score"))
.over(windowSpec).alias("highest"))
.where(col("rank") == 1)
.drop("rank")
.orderBy("department")
.show()
```

(Correct)

○

```
peopleDF
.withColumn("score", explode(col("score")))
.orderBy("department", "score")
.select(col("name"), col("department"),
first(col("score")).as("highest"))
.show()
```

◉

```
 peopleDF
.withColumn("score", explode(col("score")))
.groupBy("department")
.max("score")
.withColumnRenamed("max(score)", "highest")
.orderBy("department")
.show()
```

(Incorrect)

Question 29: **Correct**

When joining two dataframes, if there is a need to evaluate the keys in both of the DataFrames or tables and include all rows from the left DataFrame as well as any rows in the right DataFrame that have a match in the left DataFrame also If there is no equivalent row in the right DataFrame, we want to instert null: which join type we should select ?

```
df1.join(person, joinExpression, joinType)
```

- ◯ **joinType = "left_semi"**
- ◯ **joinType = "leftAnti"**
- ◉ **joinType = "left_outer"** (Correct)
- ◯ **joinType = "leftOuter"**

Question 30: **Incorrect**

Spark dynamically handles skew in sort-merge join by splitting (and replicating if needed) skewed partitions. Which property need to be enabled to achieve this ?

- ◉ **spark.sql.adaptive.skewJoin.enable** (Incorrect)
- ◯ **spark.sql.adaptive.skewJoin.enabled** (Correct)
- ◯ **spark.sql.skewJoin.enabled**
- ◯ **spark.sql.adaptive.optimeze.skewJoin**

Explanation

see https://spark.apache.org/docs/latest/sql-performance-tuning.html

Question 31: **Correct**

Your application on production is crashing lately and your application gets stuck at the same level every time you restart the spark job . You know that it is the **toLocalIterator** function is causing the problem. What are the possible solutions to this problem ?

- (●) **Reduce the size of your partitions if possible.** (Correct)
- ( ) **Reduce the memory of the driver**
- ( ) **Use collect function instead of to localIterator**
- ( ) **There is nothing to worry, application crashes are expected and will not affect your application at all.**

Question 32: **Correct**

We want to create a dataframe with a schema. Choose the correct order in order to achieve this goal.

```
1. schema = "INTEGER"
2. a = [1002, 3001, 4002, 2003, 2002, 3004, 1003, 4006]
3. spark.createDataFrame(data,schema)
4. spark.createDataSet(data, schema)
5. spark.create(data, schema)
6. spark.createDataFrame(schema, data)
```

- ( ) **1, 2, 5**
- ( ) **1, 2, 4**
- ( ) **1, 2, 6**
- (●) **1, 2, 3** (Correct)

Question 33: **Correct**

If spark is running in client mode, which of the following statement about is correct ?

- ◯ **Spark driver is attributed to the machine that has the most resources**
- ◯ **The entire spark application is run on a single machine.**
- ◉ **Spark driver remains on the client machine that submitted the application** (Correct)
- ◯ **Spark driver is randomly attributed to a machine in the cluster**

Question 34: **Correct**

You have a need to transform a column named 'date' to a timestamp format. Assume that the column 'date' is timestamp compatible. You have written the code block down below, but it contains an error. Identify and fix it.

```
df.select(to_timestamp(col("date")).show()
```

- ◯ **to_timestamp() is not a valid operation. Proper function is toTimestamp()** `df.select(toTimestamp(col("date")))`
- ◉ **Query doesn't contain an error. Default format is** `YYYY-mm-dd HH:MM:ss.SSS` (Correct)
- ◯ **We need to add a format and it should be the first parameter passed to this function.** `df.select(to_timestamp('yyyy-dd-MM', col("date")))`
- ◯ **to_timestamp() is not a valid operation. Proper function is toTimestamp() and also we need to add a format.** `df.select(toTimestamp(col("date"), 'yyyy-dd-MM')))`
- ◯ **to_timestamp requires always a format ! So you need to add one** `df.select(to_timestamp(col("date"), 'yyyy-dd-MM'))`

Question 35: **Correct**

Determine if the following statement is true or false.

When using DataFrame.persist() data on disk is always serialized.

- ○ **FALSE**
- ◉ **TRUE** (Correct)

Explanation

Data on disk is always serialized using either Java or Kryo serialization.

★ Question 36: **Correct**

Which of the following describes a worker node ?

- ○ **Worker nodes are synonymous with executors.**
- ◉ **Worker nodes are the nodes of a cluster that perform computations.** (Correct)
- ○ **Worker nodes are the most granular level of execution in the Spark execution hierarchy.**
- ○ **Worker nodes always have a one-to-one relationship with executors.**

Explanation
The role of worker nodes/executors:

1. Perform the data processing for the application code
2. Read from and write the data to the external sources
3. Store the computation results in memory, or disk.

The executors run throughout the lifetime of the Spark application. This is a static allocation of executors. The user can also decide how many numbers of executors are required to run the tasks, depending on the workload. This is a dynamic allocation of executors.

Before the execution of tasks, the executors are registered with the driver program through the cluster manager, so that the driver knows how many numbers of executors are running to perform the scheduled tasks. The executors then start executing the tasks scheduled by the worker nodes through the cluster manager.

Whenever any of the worker nodes fail, the tasks that are required to be performed will be automatically allocated to any other worker nodes

Question 37: **Correct**

Which of the following transformation is not evaluated lazily ?

- ◯ **sample()**
- ◯ **repartition()**
- ◯ **filter()**
- ◯ **select()**
- ◉ **None of the responses, all transformations are lazily evaluated.** **(Correct)**

Explanation
All transformations are lazily evaluated in spark.

Question 38: **Correct**

What causes a stage boundary ?

- ◉ **Shuffle** **(Correct)**
- ◯ **Failure of network**
- ◯ **Failure of worker node**
- ◯ **Failure of driver node**

Explanation
Not all Spark operations can happen in a single stage, they are divided into multiple stages when there is a shuffle. And this causes a stage boundary.

Question 39: **Correct**

The code block shown below contains an error. Identify the error.

```
def squared(s):
  return s * s

spark.udf.register("square", squared)
spark.range(1, 20).createOrReplaceTempView("test")
spark.sql("select id, squared(id) as id_squared from test")
```

- ◯ **We need to add quotes when using udf in sql. Proper usage should be:** `spark.sql("select id, "squared(id)" as id_squared from test")`
- ◯ **There is no column id created in the database.**
- ◉ **We need to use function 'square' instead of 'squared' in the sql command. Proper command should be:** `spark.sql("select id, square(id) as id_squared from test")` **(Correct)**
- ◯ **There is no error in the code.**
- ◯ **We are not referring to right database. Proper command should be:** `spark.sql("select id, squared(id) as id_squared from temp_test")`

Question 40: **Correct**

What is the correct syntax to run sql queries programmaticaly ?

- ◯ **It is not possible to run sql queries programmatically**
- ◉ **spark.sql()** **(Correct)**
- ◯ **spark.query()**
- ◯ **spark.run()**
- ◯ **spark.runSql()**

Question 41: **Correct**

Which of the followings are useful use cases of spark ?

- (●) **All of the answers are correct.** (Correct)
- ( ) **Building, training, and evaluating machine learning models using MLlib**
- ( ) **Performing ad hoc or interactive queries to explore and visualize data sets**
- ( ) **Analyzing graph data sets and social networks**
- ( ) **Processing in parallel large data sets distributed across a cluster**

Question 42: **Correct**

Given an instance of SparkSession named spark, reviewing the following code what's the output ?

```
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import col
import pyspark.sql.functions as f


a = [1002, 3001, 4002, 2003, 2002, 3004, 1003, 4006]
b = spark
.createDataFrame(a, IntegerType())
.withColumn("x", col("value") % 1000)

c = b
.groupBy(col("x"))
.agg(f.count("x"), f.sum("value"))
.drop("x")
.toDF("count", "total")
.orderBy(col("count").desc(), col("total"))
.limit(1)
.show()
```

○
```
|count|total|
|    8|20023|
```

○
```
|count|total|
|    1| 3001|
```

◉
```
|count|total|
|    3| 7006|
```
(Correct)

○
```
|count|total|
|    2| 8008|
```

Question 43: **Correct**

For the following dataframe if we want to fully cache the dataframe, what functions should we call in order ?

```
df = spark.range(1 * 10000000).toDF("id")
```

- [ ] Only `df.count()`
- [ ] Only `df.cache()`
- [x] `df.cache()` and then `df.count()` (Correct)
- [ ] `df.cache()` and then `df.take(1)`
- [ ] `df.take(1)`

Question 44: **Correct**

If spark is running in cluster mode, which of the following statements about nodes is incorrect ?

- ○ Each executor is running in a JVM inside of a worker node
- ○ There is at least one worker node in the cluster
- ○ There might be more executors than total number of nodes
- ◉ The spark driver runs in its own non-worker node without any executors (Correct)
- ○ There is one single worker node that contains the Spark driver and the executors

Question 45: **Correct**

At which stage do the first set of optimizations take place?

- ◯ **Physical Planning**
- ◯ **Analysis**
- ◯ **Code Generation**
- ◉ **Logical Optimization** **(Correct)**

Explanation
First set of optimizations takes place in step logical optimization. See the link for more detail: https://databricks.com/glossary/catalyst-optimizer

Question 46: **Correct**

The code block shown below should return a new DataFrame with a new column named "casted" who's value is the long equivalent of column "a" which is a integer column also this dataframe should contain all the previously existing columns from DataFrame df. Choose the response that correclty fills in the numbered blanks within the code block to complete this task. Code block: `df._1_(_2_)`

- ◯ **1. withColumn**
  2. "casted"
  3. cast(col("a")

- ◯ **1. withColumnRenamed**
  2. casted
  3. col("a").cast("long")

- ◯ **1. withColumnRenamed**
  2. "casted"
  3. col("a").cast("long")

- ◯ **1. withColumn**
  2. "casted"
  3. cast(a)

- ◉ **1. withColumn**
  2. "casted" (Correct)
  3. col("a").cast("long")

Question 47: **Correct**

Let's suppose that we have a dataframe with a column '**today**' which has a format '**YYYY-MM-DD**'. You want to add a new column to this dataframe '**week_ago'** and you want it's value to be one week prior to column 'today'. Select the correct code block.

- (●) `df.withColumn("week_ago", date_sub(col("today"), 7))` **(Correct)**
- ( ) `df.withColumn(week_ago, date_sub(col("today"), 7))`
- ( ) `df.withColumn( date_sub(col("today"), 7), "week_ago")`
- ( ) `df.withColumn("week_ago", col("today")- 7))`
- ( ) `df.withColumn("week_ago", week_sub(col("today"), 7))`

Explanation
Date_sub and date_add are some functions that exist in the following packages org.apache.spark.sql.functions.*

Question 48: **Correct**

Select the code block which counts the number of "**quantity**" for each "**invoiceNo**" in the dataframe **df**.

- ( ) `df.groupBy(InvoiceNo).agg( expr("count(Quantity)"))`
- ( ) `df.reduceBy("InvoiceNo").agg( expr("count(Quantity)"))`
- ( ) `df.groupBy("InvoiceNo").agg( expr(count(Quantity)))`
- ( ) `df.groupBy(InvoiceNo).agg( expr(count(Quantity)))`
- (●) `df.groupBy("InvoiceNo").agg( expr("count(Quantity)"))` **(Correct)**

Question 49: **Correct**

How to make sure that dataframe df has 12 partitions given that df has 4 partitions ?

- ◯ **df.setPartitition(12)**
- ◯ **df.setPartitition()**
- ◯ **df.repartition()**
- ◉ **df.repartition(12)** (Correct)

Question 50: **Correct**

Which of the following operations can be used to create a new DataFrame with a new column and all previously existing columns from an existing DataFrame ?

- ◯ **DataFrame.drop()**
- ◯ **DataFrame.filter()**
- ◯ **DataFrame.withColumnRenamed()**
- ◯ **DdataFrame.head()**
- ◉ **DataFrame.withColumn()** (Correct)

Question 51: **Correct**

If we want to store RDD as deserialized Java objects in the JVM and if the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed also replicate each partition on two cluster nodes, which storage level we need to choose ?

- ◯ **MEMORY_AND_DISK_2_SER**
- ◉ **MEMORY_AND_DISK_2** **(Correct)**
- ◯ **MEMORY_AND_DISK**
- ◯ **MEMORY_ONLY_2**

Explanation
see https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/storage/StorageLevel.html

`StorageLevel.MEMORY_AND_DISK_2` is Same as `MEMORY_AND_DISK` storage level but replicate each partition to two cluster nodes.

Question 52: **Correct**

Which of the following code blocks changes the parquet file content given that there is already a file exist with the name that we want to write ?

- ◯ **df.save.format("parquet").mode("overwrite").option("compression", "snappy").path("path")**
- ◯ **df.write.format("parquet").option("compression", "snappy").path("path")**
- ◉ **df.write.mode("overwrite").option("compression", "snappy").save("path")** **(Correct)**

Explanation
Parquet is the default file format. If you don't include the format() method, the DataFrame will still be saved as a Parquet file.

And if the file name already exist in the path given and if you don't include option `mode("overwrite")` you will get an error.

Question 53: **Correct**

Choose the equivalent code block to:

```
df.filter(col("count") < 2)
```

Where **df** is a valid dataframe which has a column named **count**

- ○ `df.select("count < 2")`
- ○ `df.getWhere("count < 2")`
- ○ `df.where("count  is smaller then 2").show(2)`
- ◉ `df.where("count < 2")` **(Correct)**
- ○ `df.where(count < 2)`

Question 54: **Correct**

The code block shown below should return a new DataFrame with 25 percent of random records from dataframe df without replacement. Choose the response that correctly fills in the numbered blanks within the code block to complete this task.

Code block:

```
df._1_(_2_, _3_, _4_)
```

- ○ **1. random**
  2. False
  3. 0.25
  4. 5

- ○ **1. sample**
  2. False
  3. 25
  4. 5

- ○ **1. sample**
  2. True
  3. 0.25
  4. 5

- ○ **1. sample**
  2. False
  3. 0.5
  4. 25

- ○ **1. take**
  2. False
  3. 0.25
  4. 5

- ◉ **1. sample**
  2. False
  3. 0.25
  4. 5

  (Correct)

Question 55: **Correct**

Choose invalid execution mode in the following responses.

- ◯ **Cluster**
- ◯ **Local**
- ◉ **Standalone** (Correct)
- ◯ **Client**

Explanation
An execution mode gives you the power to determine where the aforementioned resources are physically located when you go to run your application. You have three modes to choose from: Cluster mode, client mode and local mode. Standalone is one of the cluster manager types.

Question 56: **Correct**

Given the following statements regarding caching:

```
1  Red: The default storage level for a DataFrame is
   StorageLevel.MEMORY_AND_DISK
2
3  Green: The DataFrame class does not have an uncache() operation
4
5  Blue: The persist() method immediately loads data from its source to
   materialize the DataFrame in cache
6
7  White: Explicit caching can decrease application performance by
   interferring with the Catalyst optimizer's ability to optimize some queries.
```

Which of these statements are TRUE?

- ◯ **Green and White**
- ◉ **Red, White and Green** (Correct)
- ◯ **Green and Blue**
- ◯ **Red, Blue, and White**

Question 57: **Incorrect**

The goal of Dynamic Partition Pruning (DPP) is to allow you to read only as much data as you need. Which property needs to be set in order to use this functionality ?

- ○ **spark.sql.dynamicPartitionPruning.optimizer.enabled**
- ○ **spark.sql.optimizer.dynamicPartitionPruning.enabled** (Correct)
- ◉ **spark.sql.dynamicPartitionPruning.enabled** (Incorrect)
- ○ **spark.sql.adaptive.dynamicPartitionPruning.enabled**

Explanation
DPP can auto-optimize your queries and make them more performant automatically. Use the diagram below and the listed steps to better understand how dynamic partition pruning works. The dimension table (on the right) is queried and filtered. A hash table is built as part of the filter query. Spark uses the result of this query (and hash table) to create a broadcast variable Then, it will broadcast the filter to each executor At runtime, Spark's physical plan is changed so that the dynamic filter is applied to the fact table. For more information; https://spark.apache.org/docs/latest/configuration.html#dynamic-allocation

Question 58: **Correct**

Which of the following code blocks concatenates two DataFrames **df1** and **df2** ?

- ○ `df1.addAll(df2)`
- ○ `df1.append(df2)`
- ○ `df1.add(df2)`
- ○ `df1.appendAll(df2)`
- ◉ `df1.union(df2)` (Correct)

Explanation
DataFrames are immutable. This means users cannot append to DataFrames because that would be changing it. To append to a DataFrame, you must union the original DataFrame along with the new DataFrame.

Question 59: **Correct**

What command we can use to get the number of partition of a dataframe named df ?

- ◯ **df.getNumPartitions()**
- ◯ **df.getPartitionSize()**
- ◯ **df.rdd.getPartitionSize()**
- ◉ **df.rdd.getNumPartitions()** **(Correct)**

Explanation
Correct answer here is df.rdd.getNumPartitions()

TESTS
Practice Test
Python
2

Question 1: **Correct**

What is the first thing to try if garbage collection is a problem ?

- ◯ **Descrease java heap space size**
- ◉ **First thing to try if garbage collection is a problem is to use serialized caching** **(Correct)**
- ◯ **Persist objects in deserialized form**

Explanation
JVM garbage collection can be a problem when you have large "churn" in terms of the RDDs stored by your program. When Java needs to evict old objects to make room for new ones, it will need to trace through all your Java objects and find the unused ones. The main point to remember here is that the cost of garbage collection is proportional to the number of Java objects, so using data structures with fewer objects (e.g. an array of Ints instead of a LinkedList) greatly lowers this cost.
https://spark.apache.org/docs/latest/tuning.html#garbage-collection-tuning

Question 2: **Correct**

Which of the following code blocks merges two DataFrames **df1** and **df2** ?

- ◯ `df1.merge(df2)`
- ◯ `df1.appendAll(df2)`
- ◯ `df1.add(df2)`
- ◯ `df1.append(df2)`
- ◉ `df1.union(df2)` **(Correct)**
- ◯ `df1.addAll(df2)`

Question 3: **Correct**

Choose the correct code block to unpersist a table named '**table**'.

- ◯ **spark.catalog.uncacheTable(table)**
- ◯ **spark.uncacheTable(table)**
- ◯ **spark.uncacheTable("table")**
- ◉ **spark.catalog.uncacheTable("table")** (Correct)

Explanation
Correct usage is spark.catalog.uncacheTable("tableName")

To remove the data from the cache, just call:

```
spark.sql("uncache table table_name")

or

spark.catalog.uncacheTable("table_name")
```

To unpersist dataframe use:

```
spark.sql("df.unpersist()")
```

Another thing to remember is when using DataFrame.persist() data on disk is always serialized.

Question 4: **Correct**

The code block down below intends to join **df1** with **df2** with inner join but it contains an error. Identify the error. `d1.join(d2, d1.col(id) == df2.col(id), inner)`

- ◯ **We cannot do inner join in spark 3.0, but it is in the roadmap.**
- ◯ **The order is not correct. It should be like the following**
  `df1.join(d2, "inner", d1.col("id") === df2.col("id"))`
- ◉ **Quotes are missing. The correct query should be**
  `d1.join(d2, d1.col("id") == df2.col("id"), "inner")` (Correct)
- ◯ **There should be two === instead of ==. So the correct query is**

```
d1.join(d2, d1.col("id") === df2.col("id"), inner)
```

Question 5: Incorrect

Which of the followings are true for driver ?

- [ ] **Executing code assigned to it**
- [ ] **Runs your main() function** (Correct)
- [x] **Responsible for assigning work that will be completed in parallel.** (Correct)
- [x] **Is responsible for maintaining information about the Spark Application** (Correct)
- [ ] **Controls physical machines and allocates resources to Spark Applications**

Explanation
The driver is the machine in which the application runs. It is responsible for three main things: 1) Maintaining information about the Spark Application, 2) Responding to the user's program, 3) Analyzing, distributing, and scheduling work across the executors.

Question 6: Correct

Choose the correct code block to broadcast **dfA** and join it with **dfB** ?

- ○ `dfA.join(broadcast(dfB), dfA.id == dfB.id)`
- ○ `dfA.join(broadcast("dfB"), dfA.id == dfB.id)`
- ◉ `dfB.join(broadcast(dfA), dfA.id == dfB.id)` (Correct)
- ○ `dfB.join(broadcast("dfA"), dfA.id == dfB.id)`

Question 7: **Correct**

What will cause a full shuffle knowing that dataframe '**df**' has 2 partitions ?

- (●) `df.repartition(12)` **(Correct)**
- ( ) `df.coalse(4)`
- ( ) **All of them will cause a full shuffle.**

Explanation

Coalse function avoids a full shuffle if it's known that the number is decreasing then the executor can safely keep data on the minimum number of partitions, only moving the data off the extra nodes, onto the nodes that we kept. And it cannot be used to increase the number of partitions.

Question 8: **Correct**

Given an instance of SparkSession named spark, and the following DataFrame named **df**:

```
simpleData = [("James","Sales","NY",90000,34,10000),
    ("Michael","Sales","NY",86000,56,20000),
    ("Robert","Sales","CA",81000,30,23000),
    ("Maria","Finance","CA",90000,24,23000),
    ("Raman","Finance","CA",99000,40,24000),
    ("Scott","Finance","NY",83000,36,19000),
    ("Jen","Finance","NY",79000,53,15000),
    ("Jeff","Marketing","CA",80000,25,18000),
    ("Kumar","Marketing","NY",91000,50,21000)
  ]
schema = ["employee_name","department","state","salary","age","bonus"]
df = spark.createDataFrame(data=simpleData, schema = schema)
```

Choose the right code block which will produce the following result:

```
+----------+-----------+
|department|sum(salary)
|
+----------+-----------+
Sales     |257000
Finance   |351000
Marketing |171000
```

- ( ) `df.groupBy("department").sumAll("salary").show(truncate=False)`
- (●) `df.groupBy("department").sum("salary").show(truncate=False)` **(Correct)**
- ( ) `df.reduce("department").sum("salary").show(truncate=False)`
- ( ) `df.groupBy("department").agg("salary").show(truncate=False)`
- ( ) `df.groupBy(department).sum(salary).show(truncate=False)`

Question 9: **Correct**

How make sure that dataframe **df** has 8 partitions given that it has 4 partitions ?

- (●) `df.repartition(8)` **(Correct)**
- ( ) `df.setPartitition(8)`
- ( ) `df.coalesce(8)`
- ( ) `df.partition(8)`

Question 10: **Correct**

If we want to store RDD as serialized Java objects in the JVM and if the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed, which storage level we need to choose ?

- ( ) **MEMORY_AND_DISK_2**
- ( ) **MEMORY_AND_DISK**
- ( ) **MEMORY_ONLY_2**
- (●) **MEMORY_AND_DISK_SER** **(Correct)**

Explanation

https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.StorageLevel.html#pyspark.StorageLevel

"Since the data is always serialized on the Python side, all the constants use the serialized formats."

Question 11: **Incorrect**

The code block shown below contains an error. The code block is intended to write a text file in the path. What should we add to part 1 in order to fix ?

```
df = spark.createDataFrame([

['John','NYC'],

['Kevin','Chicago']],('name','city'))

(1)

df.write.text("my_file.txt")
```

- ○ `df.coalse(4)`
- ○ `df.take()`
- ○ `df.drop("name")` (Correct)
- ◉ `df.repartition(8)` (Incorrect)

Explanation
When you write a text file, you need to be sure to have only one string column; otherwise, the write will fail:

Question 12: **Correct**

The code block shown below should return a new DataFrame with 25 percent of random records from dataframe df with replacement. Choose the response that correctly fills in the numbered blanks within the code block to complete this task.

Code block:

df._1_(_2_, _3_, _4_)

- ○ **1. sample**
  2. True
  3. 0.5
  4. 25
- ○ **1. sample**
  2. withReplacement
  3. 0.25
  4. 5
- ◉ **1. sample**
  2. True
  3. 0.25
  4. 5 (Correct)

Question 13: **Correct**

What does the following property achieves in spark when enabled ?

```
spark.sql.adaptive.skewJoin.enabled
```

- ◯ **The goal of this property is to allow you to read only as much data as you need.**
- ◯ **It allows you to dynamically select physical plans based on cost.**
- ◯ **It allows you to dynamically convert physical plans to RDDs**
- ◉ **Spark dynamically handles skew in sort-merge join by splitting (and replicating if needed) skewed partitions with this property enabled.** (Correct)

Question 14: **Correct**

If spark is running in client mode, which of the following statement about is **NOT** correct ?

- ◯ **In this mode worker nodes reside in the cluster.**
- ◉ **The entire spark application is run on a single machine.** (Correct)
- ◯ **Machines that who runs the driver called gateway machines or edge nodes.**
- ◯ **Spark driver remains on the client machine that submitted the application.**

Explanation

Client mode is nearly the same as cluster mode except that the Spark driver remains on the client machine that submitted the application.

Question 15: **Correct**

Consider following dataframe

```
df = spark.createDataFrame([  ['John','NYC'],  ['Kevin','Chicago'],
['Ram','Delhi'],  ['Sanjay','Sdyney'],  ['Ali','Istanbul'],
['Zakaria','Paris'],  ['Alice','Chicago'],  ['Ann','Miami'],
['Hajar','Casablanca'],  ['Cassandra','Marseille']],('name','city'))
df = df.repartition(8)
```

And we apply this code block `df.rdd.getNumPartitions()` What we will see ?

- ◯ **4**
- ◯ **It is not a valid command, we will have an error**
- ◉ **8** **(Correct)**
- ◯ **1**

Question 16: **Incorrect**

Given the following statements regarding caching:

```
1: The default storage level for a DataFrame is StorageLevel.MEMORY

2: The DataFrame class does have an unpersist() operation

3: The persist() method needs an action to load data from its source to
materialize the DataFrame in cache
```

Which one is **NOT** TRUE ?

- [ ] **1,2,3**
- [ ] **2**
- [ ] **1,3**
- [x] **1,2** **(Incorrect)**
- [ ] **1** **(Correct)**

Explanation

Default storage level is `MEMORY_AND_DISK`

https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.cache.html#pyspark.sql.DataFrame.cache

https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/Dataset.html

Dataframe has `unpersist()` function;

https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.unpersist.html#pyspark.sql.DataFrame.unpersist

The persist() method needs an action to load data from its source to materialize the DataFrame in cache (and you should be using all the partitions while doing that action)

Question 17: Incorrect

Which of the following describes the relationship between cluster managers and worker nodes?

- (●) **A cluster manager is a Java Virtual Machine (JVM) running on a worker node.** (Incorrect)
- ( ) **There are always the same number of cluster managers and worker nodes.**
- ( ) **A worker node is a Java Virtual Machine (JVM) running on an cluster manager.**
- ( ) **There are always more cluster manager nodes than worker nodes.**
- ( ) **Cluster manager creates worker nodes and allocates resource to it.** (Correct)

Explanation
An executor is a Java Virtual Machine (JVM) running on a worker node. See the componenets here: https://spark.apache.org/docs/latest/cluster-overview.html

Question 18: Correct

The code block shown below should return a new DataFrame with a new column named "**casted**" who's value is the string equivalent of column "**a**" which is a integer column, also this dataframe should contain all the previously existing columns from DataFrame df. Choose the response that correctly fills in the numbered blanks within the code block to complete this task.

Code block:

```
df._1_(_2_, _3_)
```

- (●) **1. withColumn**
  2. "casted" (Correct)
  3. col("a").cast("String")
- ( ) **1. withColumnRenamed**
  2. casted
  3. col("a").cast("String")
- ( ) **1. withColumnRenamed**
  2. "casted"
  3. col("a").cast("String")
- ( ) **1. withColumn**
  2. casted
  3. col(a).cast(String)

Question 19: **Correct**

Which of the followings is NOT a useful use cases of spark ?

- (●) **Processing in parallel small data sets distributed across a cluster** (Correct)
- ( ) **Performing ad hoc or interactive queries to explore and visualize data sets**
- ( ) **Building, training, and evaluating machine learning models using MLlib**
- ( ) **Analyzing graph data sets and social networks**

Question 20: **Incorrect**

Which of the following 3 DataFrame operations are classified as an action? Choose 3 answers:

- [ ] `limit()`
- [x] `show()` (Correct)
- [ ] `foreach()` (Correct)
- [ ] `cache()`
- [x] `first()` (Correct)
- [x] `printSchema()` (Incorrect)

Explanation

see https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

Question 21: **Correct**

What happens at a stage boundary in spark ?

- ◯ **Application stops immediately .**
- ◯ **Worker nodes restarts.**
- ◯ **Stage gets transformed to a job.**
- ◉ **At each stage boundary, data is written to disk by tasks in the parent stages and then fetched over the network by tasks in the child stage.** (Correct)

Explanation

At each stage boundary, data is written to disk by tasks in the parent stages and then fetched over the network by tasks in the child stage. Because they incur heavy disk and network I/O, stage boundaries can be expensive and should be avoided when possible.

Question 22: **Correct**

Which of the following statements about Spark accumulator variables is true?

- ◉ **The Spark UI displays named accumulators used by your application.** (Correct)
- ◯ **You can define your own custom accumulator class by extending org.apache.spark.util.AccumulatorV1 in Java or Scala or pyspark.AccumulatorParams in Python.**
- ◯ **Accumulators provide a immutable variable that a Spark cluster cannot update on a per-row basis.**
- ◯ **In transformations, each task's update can be applied only once if tasks or job stages are re-executed.**
- ◯ **For accumulator restarted tasks will update the value in case of a failure.**

Question 23: Incorrect

Which of the following is true for an executor ?

- ◯ **Worker nodes are synonymous with executors.**
- ◯ **Executors nodes always have a one-to-one relationship with workers.**
- ◉ **Executors are the most granular level of execution in the Spark execution hierarchy.** (Incorrect)
- ◯ **There could be multiple exectuors in a single worker node.** (Correct)

Explanation
A worker node can be holding multiple executors (processes) if it has sufficient CPU, Memory and Storage. https://spark.apache.org/docs/latest/cluster-overview.html

Question 24: Correct

Which of the following code property is used for enabling adaptive query ?

- ◉ **spark.sql.adaptive.enabled** (Correct)
- ◯ **spark.adaptive**
- ◯ **spark.sql.optimize.adaptive**
- ◯ **spark.adaptive.sql**
- ◯ **spark.sql.adaptive**

Question 25: **Incorrect**

Which of the following three operations are classified as a wide transformation ? Choose 2 answers:

- [ ] `drop()`
- [x] `coalesce(shuffle=true)` **(Incorrect)**
- [ ] `filter()`
- [ ] `flatMap()`
- [x] `distinct()` **(Correct)**
- [x] `orderBy()` **(Correct)**

Explanation
see https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformation

Coalesce(shuffle=true) is not a valid option. You can pass number of partitions as a parameter, however it will not cause a shuffle. Check:

https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.coalesce.html

Question 26: **Correct**

For the following dataframe if we want to fully cache the dataframe immediately, what code block should replace (x) ?

```
1  df = spark.createDataFrame([  ['John','NYC','test1@mailcom'],
   ['Kevin','Chicago','test2@mail.com']],('name','city','email'))
2
3  df.cache()
4
5  (x)
```

- [ ] **df.takeAll()**
- [ ] **df.persist()**
- [ ] **df.cache()**
- [ ] **df.take(1)**
- [x] **df.count()** **(Correct)**

Question 27: **Correct**

You have a need to sort a dataframe **df** which has some null values on column **a**. You want the null values to appear last, and then the rest of the rows should be ordered ascending based on the column a. Choose the right code block to achieve your goal.

- ○ `df.sort(asc_nulls_last(a))`
- ○ **It is not possible to sort, when there are null values on the specified column.**
- ○ `df.orderBy(asc("a"))`
- ○ `df.orderBy(asc_nulls_last(a))`
- ◉ `df.orderBy(df.a.asc_nulls_last())` **(Correct)**

Question 28: **Correct**

Choose the right code block in order to change add a new column to the following schema.

```
schema = StructType([StructField("name",StringType(),True) ])
```

- ◉ `schema.add("new_column",StringType(),True)` **(Correct)**
- ○ `schema.add(StringType(), "new_column")`
- ○ `schema.append(StringType(), "new_column")`
- ○ `schema.append("new_column",StringType(),True)`

Question 29: **Correct**

There is a temp view named '**my_view**'. If I want to query this view within spark, which command I should choose ?

- ◯ `spark.read.table("global_temp.my_view")`
- ◯ `spark.read.view("my_view")`
- ◯ `spark.read.view("my_view")`
- ◉ `spark.read.table("my_view")` **(Correct)**

Explanation
Global temp views are accessed via prefix 'global_temp' And other tables are accessed without any prefixes.

Question 30: **Correct**

We have an unmanaged table "**my_table**"

If we run the code block down below

```
spark.sql("DROP TABLE IF EXISTS my_table")
```

What will happen to data in my_table ?

- ◯ **This is not a valid code block**
- ◉ **No data will be removed but you will no longer be able to refer to this data by the table name.** **(Correct)**
- ◯ **Spark will remove the table and also associated views**
- ◯ **The data will be dropped also**

Question 31: **Correct**

Your manager gave you a task to remove sensitive data. Choose the correct code block down below to remove name and city from the dataframe.

```
df = spark.createDataFrame([  ['Josh','Virginia',25,'M'],
['Adam','Paris',34,'M']],('name','city','age','gender'))
```

- (●) `df.drop("name","city")` **(Correct)**
- ( ) `df.remove("name","city")`
- ( ) `df.drop(name,city)`
- ( ) `df.remove(name,city)`
- ( ) `df.drop("Josh","Adam")`

Question 32: **Correct**

The following statement will create a managed table

```
dataframe.write.saveAsTable("unmanaged_my_table")
```

- (●) **TRUE** **(Correct)**
- ( ) **FALSE**

Explanation

One important note is the concept of managed versus unmanaged tables. Tables store two important pieces of information. The data within the tables as well as the data about the tables; that is, the metadata. You can have Spark manage the metadata for a set of files as well as for the data. When you define a table from files on disk, you are defining an unmanaged table. When you use saveAsTable on a DataFrame, you are creating a managed table for which Spark will track of all of the relevant information.

Question 33: **Correct**

When joining two dataframes, if there is a need to evaluate the keys in both of the DataFrames or tables and include all rows from the right DataFrame as well as any rows in the left DataFrame that have a match in the righ DataFrame also If there is no equivalent row in the left DataFrame, we want to instert null: which join type we should select ? `df1.join(person, joinExpression, joinType)`

- ◯ `joinType = "rightAnti"`
- ◯ `joinType = "rightOuter"`
- ◉ `joinType = "right_outer"` **(Correct)**
- ◯ `joinType = "rightSemi"`

Question 34: **Correct**

Given the following dataframe

```
df = spark.createDataFrame([  ['John','NYC'],  ['Kevin','Chicago'],
['Ram','Delhi'],  ['Sanjay','Sdyney'],  ['Ali','Istanbul'],
['Zakaria','Paris'],  ['Alice','Chicago'],  ['Ann','Miami'],
['Hajar','Casablanca'],  ['Cassandra','Marseille']],('name','city'))
df = df.repartition(8)
```

We execute the following code block

```
df.write.mode("overwrite").option("compression", "snappy").save("path")
```

Choose the correct number of files after a successful write operation.

- ◉ **8** **(Correct)**
- ◯ **4**
- ◯ **1**

Explanation

We control the parallelism of files that we write by controlling the partitions prior to writing and therefore the number of partitions before writing equals to number of files created after the write operation. If you don't specify number of partitions normally, spark tries to set the number of partitions automatically based on your cluster but here we specified that we want to have 8 partitions after we created the dataframe, so we will have 8 files in the directory.

Question 35: **Correct**

Which of the following operation is classified as a narrow transformation ?

- ○ `collect()`
- ◉ `map()` (Correct)
- ○ `distinct()`
- ○ `repartition()`
- ○ `orderBy()`

Question 36: **Correct**

Which of the following is correct for the cluster manager ?

- ○ **It is interchangeable with job.**
- ○ **Executes code assigned to it.**
- ○ **Reports the state of the computation back to the driver.**
- ◉ **Keeps track of the resources available.** (Correct)
- ○ **All of the answers are correct.**

Question 37: **Correct**

Choose valid execution modes in the following responses.

- [x] **Local** (Correct)
- [ ] **Standalone**
- [x] **Client** (Correct)
- [x] **Cluster** (Correct)

Explanation
An execution mode gives you the power to determine where the aforementioned resources are physically located when you go to run your application. You have three modes to choose from: Cluster mode, client mode and local mode. Standalone is one of the cluster manager types.

Which of the following statement is **NOT** true for broadcast variables ?

- [ ] **Broadcast variables are shared, immutable variables that are cached on every machine in the cluster instead of serialized with every single task.**
- [x] **It is a way of updating a value inside of a variety of transformations and propagating that value to the driver node in an efficient and fault-tolerant way.** (Correct)
- [ ] **You can define your own custom broadcast class by extending org.apache.spark.util.BroadcastV2 in Java or Scala or pyspark.AccumulatorParams in Python.** (Correct)
- [x] **It provides a mutable variable that a Spark cluster can safely update on a per-row basis.** (Correct)
- [ ] **The canonical use case is to pass around a small large table that does fit in memory on the executors.**

Explanation
Broadcast variables are a way you can share an immutable value efficiently around the cluster without encapsulating that variable in a function closure. The normal way to use a variable in your driver node inside your tasks is to simply reference it in your function closures (e.g., in a map operation), but this can be inefficient, especially for large variables such as a lookup table or a machine learning model. The reason for this is that when you use a variable in a closure, it must be deserialized on the worker nodes many times (one per task)

Question 39: **Correct**

Which property is used to to allocates jobs to different resource pools to achieve resources scheduling within an application ?

- ◯ **There is no need to set a property since spark is by default capable of resizing**
- ◯ **Dynamic allocation**
- ◉ **Fair Scheduler** **(Correct)**

Explanation

If you would like to run multiple Spark Applications on the same cluster, Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload. This means that your application can give resources back to the cluster if they are no longer used, and request them again later when there is demand. This feature is particularly useful if multiple applications share resources in your Spark cluster. This feature is disabled by default and available on all coarse-grained cluster managers; that is, standalone mode, YARN mode, and Mesos coarse-grained mode. There are two requirements for using this feature. First, your application must set spark.dynamicAllocation.enabled to true. Second, you must set up an external shuffle service on each worker node in the same cluster and set spark.shuffle.service.enabled to true in your application. The purpose of the external shuffle service is to allow executors to be removed without deleting shuffle files written by them. The Spark Fair Scheduler specifies resource pools and allocates jobs to different resource pools to achieve resource scheduling **within an application**. In this way, the computing resources are effectively used and the runtime of jobs is balanced, ensuring that the subsequently-submitted jobs are not affected by over-loaded jobs.

Question 40: **Correct**

What is the best description of a catalog ?

- ◯ **It is sparks core unit to parellize it's workflow.**
- ◯ **A JVM process in order to help garbage collection.**
- ◉ **It's the interface for managing a metadata catalog of relational entites such as databases, tables, functions etc.** (Correct)
- ◯ **Logically equivalent of DataFrames.**

Explanation

The highest level abstraction in Spark SQL is the Catalog. The Catalog is an abstraction for the storage of metadata about the data stored in your tables as well as other helpful things like databases, tables, functions, and views

Question 41: **Correct**

At which stage Catalyst optimizer generates one or more physical plans ?

- ◯ **Code Generation**
- ◯ **Logical Optimization**
- ◉ **Physical Planning** (Correct)
- ◯ **Analysis**

Question 42: **Correct**

Which of the following code blocks returns a DataFrame with two new columns '**a**' and '**b**' from the existing column '**aSquared**' where the values of 'a' and 'b' is half of the column 'aSquared' ?

- ○ `df.withColumn(aSquared, col(a) * col(a))`
- ○ `df.withColumn("aSquared" /2 , col(a) ).withColumn("aSquared" /2 , col(b) )`
- ○ `df.withColumn(aSquared/2 , col(a) ) .withColumn(aSquared /2 , col(b) )`
- ○ `df.withColumn("aSquared" /2 , col("a") ).withColumn("aSquared" /2 , col("b") )`
- ◉ `df.withColumn("a", col("aSquared")/2).withColumn("b", col("aSquared")/2)` **(Correct)**

Question 43: **Correct**

You have a need to transform a column named '**timestamp**' to a date format. Assume that the column 'timestamp' is compatible with format date. You have written the code block down below, but it contains an error. Identify and fix it.

```
df.select(to_date(col("timestamp"), "MM-dd-yyyy").show()
```

- ○ **to_date() is not a valid operation. Proper function is toDate() and also we need to change the format.** `df.select(toDate(col("timestamp"), "yyyy-MM-dd").show()`
- ◉ **Format is not correct. You need to change it to:** `df.select(to_date(col("timestamp"), 'yyyy-dd-MM'))` **(Correct)**
- ○ **We need to add a format and it should be the first parameter passed to this function.** `df.select(to_timestamp('yyyy-dd-MM', col("date")))`
- ○ **to_date() is not a valid operation. Proper function is toDate()** `df.select(toDate(col("timestamp"), "MM-dd-yyyy").show()`

Question 44: **Correct**

Let's suppose that we have a dataframe **df** with a column '**today**' which has a format 'YYYY-MM-DD'. You want to add a new column to this dataframe '**week_later**' and you want it's value to be one week after to column 'today'. Select the correct code block.

- ◯ `df.withColumn(week_later, date_add(col("today"), 7))`
- ◯ `df.withColumn("week_later", col("today") + 7))`
- ◯ `df.withColumn("week_later", week_add(col("today"), 7))`
- ◉ `df.withColumn("week_later", date_add(col("today"), 7))` **(Correct)**
- ◯ `df.withColumn( date_add(col("today"), 7), "week_later")`

Explanation

Date_sub and date_add are some functions that exist in the following packages org.apache.spark.sql.functions.*

Question 45: **Correct**

Given the code block down below, a database test containing nulls, identify the error.

```
def my_udf(s):
  If s is not None:
    return len(s)
  else:
    return 0

spark.udf.register("strlen", my_udf)
spark.sql("select s from test where s is not null and strlen(s) > 1")
```

- ◯ **We need to use function 'query' instead of 'sql' to query table test.**
- ◯ **This WHERE clause does not guarantee the strlen UDF to be invoked after filtering out nulls. So we will have null pointer**
- ◯ **We need to create the function first and then pass it to udf.register**
- ◉ **There are no problems in this query.** (Correct)

Explanation

Spark SQL (including SQL and the DataFrame and Dataset APIs) does not guarantee the order of evaluation of subexpressions. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order. For example, logical AND and OR expressions do not have left-to-right "short-circuiting" semantics. To perform proper null checking, we recommend that you do either of the following: Make the UDF itself null-aware and do null checking inside the UDF itself Use IF or CASE WHEN expressions to do the null check and invoke the UDF in a conditional branch

Question 46: **Correct**

Which of the following operations can be used to create a new DataFrame with only the column "**a**" from the existing DataFrame **df** ?

- ◯ `ddataFrame.head()`
- ◉ `dataFrame.select("a")` **(Correct)**
- ◯ `dataFrame.drop("a")`
- ◯ `dataFrame.withColumn()`
- ◯ `dataFrame.withColumnRenamed()`

Question 47: **Correct**

If we want to create a constant string 1 as a new column '**new_column**' in the dataframe **df**, which code block we should select ?

- ◯ `df.withColumnRenamed('new_column', lit(1))`
- ◯ `df.withColumn(new_column, lit(1))`
- ◯ `df.withColumn("new_column", 1)`
- ◯ `df.withColumn("new_column", lit(1))`
- ◉ `df.withColumn("new_column", lit("1"))` **(Correct)**

Question 48: **Correct**

Which of the following describes a stage best ?

- ◯ **User program built on Spark. Consists of a driver program and executors on the cluster.**
- ◯ **A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them.**
- ◯ **An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)**
- ◯ **A unit of work that will be sent to one executor.**
- ◉ **A physical unit of execution which is a sequence of tasks that can all be run together in parallel without a shuffle.** (Correct)

Question 49: **Correct**

We want to drop any rows that how a null value. Choose the correct order in order to achieve this goal.

```
1. df.
2. drop.
3. na()
4. drop(how='all')
5. dropna(how='any')
```

- ◯ **1, 4**
- ◉ **1, 5** (Correct)
- ◯ **1, 2, 3**
- ◯ **1, 2, 6**

Question 50: **Correct**

Which of the following code blocks changes the parquet file content given that there is already a file exist with the name that we want to write ?

- ○ `df.format(“parquet”).mode("overwrite").option("compression", "snappy").save("path")`
- ○ `df.write.format(“parquet”).option("compression", "snappy").path("path")`
- ○ `df.write.option("compression", "snappy").save("path")`
- ◉ `df.write.mode("overwrite").option("compression", "snappy").save("path")` **(Correct)**

Explanation

Parquet is the default file format. If you don't include the format() method, the DataFrame will still be saved as a Parquet file. If we don't include mode 'overwrite' our application will crash since there is already a file exist with the same name.

Question 51: **Correct**

Given the following dataframe `df = spark.createDataFrame([("A", 20), ("B", 30), ("D", 80)],["Letter", "Number"])`

We want to store the sum of all "**number**"s in a variable '**result**'. Choose the correct code block in order to achieve this goal.

- ◉ `result = df.groupBy().sum().collect()[0][0]` **(Correct)**
- ○ `result = df.reduce().sum().collect()[0][0]`
- ○ `result = df.groupBy().sum().collect()`
- ○ `result = df.groupBy().sum()`

Question 52: **Correct**

When the property `spark.sql.optimizer.dynamicPartitionPruning.enabled` is set to true, what optimization happens in spark ?

- ◯ **Spark dynamically handles skew in sort-merge join by splitting (and replicating if needed) skewed partitions with this property enabled.**
- ◯ **It allows you to dynamically switch join strategies.**
- ◉ **You to read only as much data as you need.** (Correct)
- ◯ **It allows you to dynamically select physical plans based on cost.**

Explanation
DPP can auto-optimize your queries and make them more performant automatically. For more information; https://spark.apache.org/docs/latest/configuration.html#dynamic-allocation

Question 53: **Correct**

Which of the following transformation is not evaluated lazily ?

- ◯ `filter()`
- ◯ `sample()`
- ◯ `select()`
- ◯ `repartition()`
- ◉ **None of the responses, all transformations are lazily evaluated.** (Correct)

Question 54: **Correct**

The code block shown below intends to return a new DataFrame with column "**old**" renamed to "**new**" but it contains an error. Identify the error.

```
df.withColumnRenamed(old, new)
```

- (●) **You need to add quotes; correct usage is** `df.withColumnRenamed("old", "new")` **(Correct)**
- ( ) **We need to add 'col' to specifiy that it's a column.** `df.withColumnRenamed(col("new"), col("old"))`
- ( ) **WithColumnRenamed is not a valid fonction , we need to use** `df.withColumnRenamed("new", "old")`
- ( ) **You need to reverse parameters and add quotes. So correct code block is** `df.withColumnRenamed("new", "old")`

Question 55: **Correct**

Select the code block which counts distinct number of "**quantity**" for each "**invoiceNo**" in the dataframe **df**.

- ( ) `df.reduceBy("InvoiceNo").agg( expr("count(Quantity)"))`
- ( ) `df.groupBy(InvoiceNo).agg( expr(count(Quantity)))`
- (●) `df.groupBy("InvoiceNo").agg( expr("countDisctinct(Quantity)"))` **(Correct)**
- ( ) `df.groupBy(InvoiceNo).agg( expr("count(Quantity)"))`
- ( ) `df.groupBy("InvoiceNo").agg( expr(count(Quantity)))`

Question 56: **Correct**

Choose the equivalent code block to:

```
df.filter(col("count") < 2)
```

Where df is a valid dataframe which has a column named count

- ○ `df.select("count < 2")`
- ○ `df.where(count < 2)`
- ◉ `df.where("count < 2")` **(Correct)**
- ○ `df.getWhere("count < 2")`
- ○ `df.where("count  is smaller then 2").show(2)`

Question 57: **Correct**

Consider the following DataFrame:

```
simpleData = (("James", "Sales", 3000), \
    ("Michael", "Sales", 4600),  \
    ("Robert", "Sales", 4100),   \
    ("Maria", "Finance", 3000),  \
    ("James", "Sales", 3000),    \
    ("Scott", "Finance", 3300),  \
    ("Jen", "Finance", 3900),    \
    ("Jeff", "Marketing", 3000), \
    ("Kumar", "Marketing", 2000),\
    ("Saif", "Sales", 4100) \
  )
columns= ["employee_name", "department", "salary"]
df = spark.createDataFrame(data = simpleData, schema = columns)
```

Select the code fragment that produces the following result:

```
+-------------+----------+------+----------+
|employee_name|department|salary|dense_rank|
+-------------+----------+------+----------+ |
 James|     Sales|  3000|         1|
 James|     Sales|  3000|         1|
 Robert|     Sales|  4100|         2|
 Saif|     Sales|  4100|         2|
 Michael|     Sales|  4600|         3|
 Maria|   Finance|  3000|         1|
 Scott|   Finance|  3300|         2|
 Jen|   Finance|  3900|         3|
 Kumar| Marketing|  2000|         1|
 Jeff| Marketing|  3000|         2|
```

- ○ **windowSpec = Window.partitionBy("department").orderBy("salary")**
  df.withColumn("rank",rank().over(windowSpec)).show()

- ○ **windowSpec = Window.partitionBy("department").orderBy("name")**
  df.withColumn("rank",rank().over(windowSpec)) .show()

- ◉ **from pyspark.sql.functions import dense_rank**
  windowSpec =
  Window.partitionBy("department").orderBy("salary") (Correct)
  df.withColumn("dense_rank",dense_rank().over(windowSpec))
  .show()

- ○ **windowSpec = Window.partitionBy("department").orderBy("salary")**
  df.withColumn("rank",rank().over("windowSpec")) .show()

- ○ **from pyspark.sql.functions import dense_rank windowSpec =**
  Window.partitionBy("department").orderBy("salary")
  df.withColumn("dense_rank",dense_rank().over("windowSpec")) .show()

Question 58: **Correct**

If spark is running in cluster mode, which of the following statements about nodes is correct ?

- ◯ **There is always more than one node.**
- ◯ **Each executor is running JVM inside of a cluster manager node.**
- ◯ **There are less executors than total number of nodes**
- ◯ **There is one single worker node that contains the Spark driver and all the executors.**
- ◉ **The spark driver runs in worker node inside the cluster.** (Correct)

Explanation

In cluster mode, a user submits a pre-compiled JAR, Python script, or R script to a cluster manager. The cluster manager then launches the driver process on a worker node inside the cluster, in addition to the executor processes.

Question 59: **Correct**

The code block shown below contains an error. Identify the error.

```
  def squared (s):
    return s*s

 spark.udf.register("square", squared) spark.range(1,
20).createOrReplaceTempView("test") spark.sql(“select id, square(id) as
id_squared from temp_test”)
```

- ◯ **There is no column id created in the database.**
- ◯ **There is no error in the code.**
- ◉ **We are not querying the right view. Correct code block should be:** (Correct)
  `spark.sql(“select id, square(id) as id_squared from test”)`
- ◯ **We need to add quotes when using udf in sql. Proper usage should be:**
  `spark.sql(“select id, “squared(id)” as id_squared from test”)`
- ◯ **We are not refering to right database. Proper command should be:**
  `spark.sql(“select id, squared(id) as id_squared from temp_test”)`

Question 60: **Correct**

Which of the following code blocks reads from a tsv file where values are separated with '`\t`'?

- `spark.read.option("header", "true").option(“inferSchema”, “true”).option(“sep”, “true”).toDf(file)`
- `spark.read.format("tsv").option("header", "true").option(“inferSchema”, “true”).load(file)`
- (selected) `spark.read.format("csv").option("header", "true").option(“inferSchema”, “true”).option(“sep”, “\t”).load(file)` **(Correct)**
- `spark.load.option("header", "true").option(“inferSchema”, “true”).read(file)`

Explanation

With Spark 2.0+ we can use CSV connector to read a tsv file.

Databricks Pyspark Demo Practice Test

Question 1

Which of the following statements about the Spark driver is incorrect?

A. The Spark driver is the node in which the Spark application's main method runs to coordinate the Spark application.
B. The Spark driver is horizontally scaled to increase overall processing throughput.
C. The Spark driver contains the SparkContext object.
D. The Spark driver is responsible for scheduling the execution of data by various worker nodes in cluster mode.
E. The Spark driver should be as close as possible to worker nodes for optimal performance.

Question 2

Which of the following describes nodes in cluster-mode Spark?

A. Nodes are the most granular level of execution in the Spark execution hierarchy.
B. There is only one node and it hosts both the driver and executors.
C. Nodes are another term for executors, so they are processing engine instances for performing computations.
D. There are driver nodes and worker nodes, both of which can scale horizontally.
E. Worker nodes are machines that host the executors responsible for the execution of tasks.

Question 3

Which of the following statements about slots is true?

A. There must be more slots than executors.
B. There must be more tasks than slots.
C. Slots are the most granular level of execution in the Spark execution hierarchy.
D. Slots are not used in cluster mode.
E. Slots are resources for parallelization within a Spark application.

Question 4

Which of the following is a combination of a block of data and a set of transformers that will run on a single executor?

A. Executor
B. Node
C. Job
D. Task
E. Slot

Question 5

Which of the following is a group of tasks that can be executed in parallel to compute the same set of operations on potentially multiple machines?

A. Job
B. Slot
C. Executor
D. Task
E. Stage

Question 6

Which of the following describes a shuffle?

A. A shuffle is the process by which data is compared across partitions.
B. A shuffle is the process by which data is compared across executors.
C. A shuffle is the process by which partitions are allocated to tasks.
D. A shuffle is the process by which partitions are ordered for write.
E. A shuffle is the process by which tasks are ordered for execution.

Question 7

DataFrame df is very large with a large number of partitions, more than there are executors in the cluster. Based on this situation, which of the following is incorrect? Assume there is one core per executor.

A. Performance will be suboptimal because not all executors will be utilized at the same time.
B. Performance will be suboptimal because not all data can be processed at the same time.
C. There will be a large number of shuffle connections performed on DataFrame df when operations inducing a shuffle are called.
D. There will be a lot of overhead associated with managing resources for data processing within each task.
E. There might be risk of out-of-memory errors depending on the size of the executors in the cluster.

Question 8

Which of the following operations will trigger evaluation?

A. `DataFrame.filter()`
B. `DataFrame.distinct()`
C. `DataFrame.intersect()`
D. `DataFrame.join()`
E. `DataFrame.count()`

Question 9

Which of the following describes the difference between transformations and actions?

A. Transformations work on DataFrames/Datasets while actions are reserved for native language objects.
B. There is no difference between actions and transformations.
C. Actions are business logic operations that do not induce execution while transformations are execution triggers focused on returning results.
D. Actions work on DataFrames/Datasets while transformations are reserved for native language objects.
E. Transformations are business logic operations that do not induce execution while actions are execution triggers focused on returning results.

Question 10

Which of the following DataFrame operations is always classified as a narrow transformation?

A. `DataFrame.sort()`
B. `DataFrame.distinct()`
C. `DataFrame.repartition()`
D. `DataFrame.select()`
E. `DataFrame.join()`

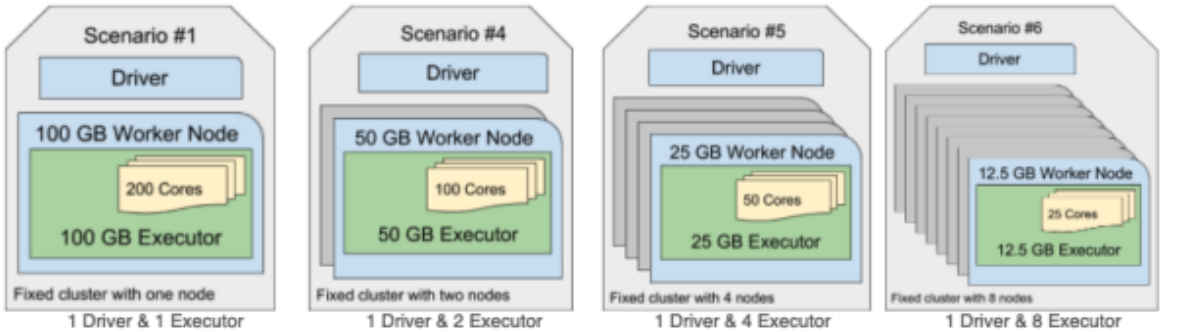
Question 11

Spark has a few different execution/deployment modes: cluster, client, and local. Which of the following describes Spark's execution/deployment mode?

A. Spark's execution/deployment mode determines where the driver and executors are physically located when a Spark application is run
B. Spark's execution/deployment mode determines which tasks are allocated to which executors in a cluster
C. Spark's execution/deployment mode determines which node in a cluster of nodes is responsible for running the driver program
D. Spark's execution/deployment mode determines exactly how many nodes the driver will connect to when a Spark application is run
E. Spark's execution/deployment mode determines whether results are run interactively in a notebook environment or in batch

Question 12

Which of the following cluster configurations will ensure the completion of a Spark application in light of a worker node failure?



Note: each configuration has roughly the same compute power using 100GB of RAM and 200 cores.

A. Scenario #1
B. They should all ensure completion because worker nodes are fault-tolerant.
C. Scenario #4
D. Scenario #5
E. Scenario #6

Question 13

Which of the following describes out-of-memory errors in Spark?

A. An out-of-memory error occurs when either the driver or an executor does not have enough memory to collect or process the data allocated to it.

B. An out-of-memory error occurs when Spark's storage level is too lenient and allows data objects to be cached to both memory and disk.
C. An out-of-memory error occurs when there are more tasks than are executors regardless of the number of worker nodes.
D. An out-of-memory error occurs when the Spark application calls too many transformations in a row without calling an action regardless of the size of the data object on which the transformations are operating.
E. An out-of-memory error occurs when too much data is allocated to the driver for computational purposes.

Question 14

Which of the following is the default storage level for `persist()` for a non-streaming DataFrame/Dataset?

A. `MEMORY_AND_DISK`
B. `MEMORY_AND_DISK_SER`
C. `DISK_ONLY`
D. `MEMORY_ONLY_SER`
E. `MEMORY_ONLY`

Question 15

Which of the following describes a broadcast variable?

A. A broadcast variable is a Spark object that needs to be partitioned onto multiple worker nodes because it's too large to fit on a single worker node.
B. A broadcast variable can only be created by an explicit call to the broadcast() operation.
C. A broadcast variable is entirely cached on the driver node so it doesn't need to be present on any worker nodes.
D. A broadcast variable is entirely cached on each worker node so it doesn't need to be shipped or shuffled between nodes with each stage.
E. A broadcast variable is saved to the disk of each worker node to be easily read into memory when needed.

Question 16

Which of the following operations is most likely to induce a skew in the size of your data's partitions?

A. `DataFrame.collect()`
B. `DataFrame.cache()`
C. `DataFrame.repartition(n)`
D. `DataFrame.coalesce(n)`
E. `DataFrame.persist()`

Question 17

Which of the following data structures are Spark DataFrames built on top of?

A. Arrays
B. Strings
C. RDDs
D. Vectors
E. SQL Tables

Question 18

Which of the following code blocks returns a DataFrame containing only column **`storeId`** and column **`division`** from DataFrame **`storesDF`**?

A. `storesDF.select("storeId").select("division")`
B. `storesDF.select(storeId, division)`
C. `storesDF.select("storeId", "division")`
D. `storesDF.select(col("storeId", "division"))`
E. `storesDF.select(storeId).select(division)`

Question 19

Which of the following code blocks returns a DataFrame containing all columns from DataFrame **`storesDF`** except for column sqft and column **`customerSatisfaction`**?

A sample of DataFrame **`storesDF`** is below:

storeId	open	openDate	division	sqft	numberOfEmployees	customerSatisfaction
0	true	1100746394	Utah	43161	61	71.1
1	true	944572255	West Virginia	18132	96	43.46
2	false	925495628	West Virginia	79520	45	36.93
3	true	1397353092	Texas	47751	78	47.19
4	true	986505057	Delaware	81483	95	25.24
...	...	...	...	...	...	...

A. `storesDF.drop("sqft", "customerSatisfaction")`
B. `storesDF.select("storeId", "open", "openDate", "division")`
C. `storesDF.select(-col(sqft), -col(customerSatisfaction))`
D. `storesDF.drop(sqft, customerSatisfaction)`
E. `storesDF.drop(col(sqft), col(customerSatisfaction))`

Question 20

The below code shown block contains an error. The code block is intended to return a DataFrame containing only the rows from DataFrame **storesDF** where the value in DataFrame **storesDF**'s "sqft" column is less than or equal to 25,000. Assume DataFrame **storesDF** is the only defined language variable. Identify the error.

Code block:

```
storesDF.filter(sqft <= 25000)
```

A. The column name **sqft** needs to be quoted like **storesDF.filter("sqft" <= 25000)**.
B. The column name **sqft** needs to be quoted and wrapped in the **col()** function like **storesDF.filter(col("sqft") <= 25000)**.
C. The sign in the logical condition inside **filter()** needs to be changed from **<=** to **>**.
D. The sign in the logical condition inside **filter()** needs to be changed from **<=** to **>=**.
E. The column name sqft needs to be wrapped in the **col()** function like **storesDF.filter(col(sqft) <= 25000)**.

Question 21

The code block shown below should return a DataFrame containing only the rows from DataFrame **storesDF** where the value in column **sqft** is less than or equal to 25,000 OR the value in column **customerSatisfaction** is greater than or equal to 30. Choose the response that correctly fills in the numbered blanks within the code block to complete this task.

Code block:

```
storesDF.__1__(__2__ __3__ __4__)
```

A.

```
1. filter
2. (col("sqft") <= 25000)
3. |
4. (col("customerSatisfaction") >= 30)
```

B.

```
1. drop
2. (col(sqft) <= 25000)
3. |
4. (col(customerSatisfaction) >= 30)
```

C.

```
1. filter
2. col("sqft") <= 25000
3. |
4. col("customerSatisfaction") >= 30
```

D.

```
1. filter
2. col("sqft") <= 25000
3. or
4. col("customerSatisfaction") >= 30
```

E.

```
1. filter
2. (col("sqft") <= 25000)
3. or
4. (col("customerSatisfaction") >= 30)
```

Question 22

Which of the following operations can be used to convert a DataFrame column from one type to another type?

A. `col().cast()`
B. `convert()`
C. `castAs()`
D. `col().coerce()`
E. `col()`

Question 23

Which of the following code blocks returns a new DataFrame with a new column **`sqft100`** that is 1/100th of column **`sqft`** in DataFrame **`storesDF`**? Note that column **`sqft100`** is not in the original DataFrame **`storesDF`**.

A. `storesDF.withColumn("sqft100", col("sqft") * 100)`
B. `storesDF.withColumn("sqft100", sqft / 100)`
C. `storesDF.withColumn(col("sqft100"), col("sqft") / 100)`
D. `storesDF.withColumn("sqft100", col("sqft") / 100)`
E. `storesDF.newColumn("sqft100", sqft / 100)`

Question 24

Which of the following code blocks returns a new DataFrame from DataFrame **`storesDF`** where column **`numberOfManagers`** is the constant integer **`1`**?

A. `storesDF.withColumn("numberOfManagers", col(1))`
B. `storesDF.withColumn("numberOfManagers", 1)`
C. `storesDF.withColumn("numberOfManagers", lit(1))`
D. `storesDF.withColumn("numberOfManagers", lit("1"))`
E. `storesDF.withColumn("numberOfManagers", IntegerType(1))`

Question 25

The code block shown below contains an error. The code block intends to return a new DataFrame where column **`storeCategory`** from DataFrame **`storesDF`** is split at the underscore character into column **`storeValueCategory`** and column **`storeSizeCategory`**. Identify the error.

A sample of DataFrame **`storesDF`** is displayed below:

storeId	open	openDate	storeCategory
0	true	1100746394	VALUE_MEDIUM
1	true	944572255	MAINSTREAM_SMALL
2	false	925495628	PREMIUM_LARGE
3	true	1397353092	VALUE_MEDIUM
4	true	986505057	VALUE_LARGE
5	true	955988614	PREMIUM_LARGE
...	...	...	...

Code block:

```
(storesDF.withColumn(
        "storeValueCategory", col("storeCategory").split("_")[0]
    ).withColumn(
        "storeSizeCategory", col("storeCategory").split("_")[1]
    )
)
```

A. The **`split()`** operation comes from the imported functions object. It accepts a string column name and split character as arguments. It is not a method of a Column object.

B. The **`split()`** operation comes from the imported functions object. It accepts a Column object and split character as arguments. It is not a method of a Column object.

C. The index values of 0 and 1 should be provided as second arguments to the **`split()`** operation rather than indexing the result.

D. The index values of 0 and 1 are not correct — they should be 1 and 2, respectively.

E. The **`withColumn()`** operation cannot be called twice in a row.

Question 26

Which of the following operations can be used to split an array column into an individual DataFrame row for each element in the array?

A. **`extract()`**
B. **`split()`**
C. **`explode()`**
D. **`arrays_zip()`**
E. **`unpack()`**

Question 27

Which of the following code blocks returns a new DataFrame where column **`storeCategory`** is an all-lowercase version of column **`storeCategory`** in DataFrame **`storesDF`**? Assume DataFrame **`storesDF`** is the only defined language variable.

A.
```
storesDF.withColumn("storeCategory",
lower(col("storeCategory")))
```
B.
```
storesDF.withColumn("storeCategory",
col("storeCategory").lower())
```
C.
```
storesDF.withColumn("storeCategory",
tolower(col("storeCategory")))
```
D.
```
storesDF.withColumn("storeCategory", lower("storeCategory"))
```
E.
```
storesDF.withColumn("storeCategory", lower(storeCategory))
```

Question 28

The code block shown below contains an error. The code block is intended to return a new DataFrame where column **`division`** from DataFrame **`storesDF`** has been renamed to column **`state`** and column **`managerName`** from DataFrame **`storesDF`** has been renamed to column **`managerFullName`**. Identify the error.

Code block:

```
(storesDF.withColumnRenamed("state", "division")
         .withColumnRenamed("managerFullName", "managerName"))
```

A. Both arguments to operation **`withColumnRenamed()`** should be wrapped in the **`col()`** operation.
B. The operations **`withColumnRenamed()`** should not be called twice, and the first argument should be **`["state", "division"]`** and the second argument should be **`["managerFullName", "managerName"]`**.
C. The old columns need to be explicitly dropped.
D. The first argument to operation **`withColumnRenamed()`** should be the old column name and the second argument should be the new column name.
E. The operation **`withColumnRenamed()`** should be replaced with **`withColumn()`**.

Question 29

Which of the following code blocks returns a DataFrame where rows in DataFrame **`storesDF`** containing missing values in every column have been dropped?

A. `storesDF.nadrop("all")`
B. `storesDF.na.drop("all", subset = "sqft")`
C. `storesDF.dropna()`
D. `storesDF.na.drop()`
E. `storesDF.na.drop("all")`

Question 30

Which of the following operations fails to return a DataFrame where every row is unique?

A. `DataFrame.distinct()`
B. `DataFrame.drop_duplicates(subset = None)`
C. `DataFrame.drop_duplicates()`
D. `DataFrame.dropDuplicates()`
E. `DataFrame.drop_duplicates(subset = "all")`

Question 31

Which of the following code blocks will not always return the exact number of distinct values in column division?

A. `storesDF.agg(approx_count_distinct(col("division")).alias("divisionDistinct"))`
B. `storesDF.agg(approx_count_distinct(col("division"), 0).alias("divisionDistinct"))`
C. `storesDF.agg(countDistinct(col("division")).alias("divisionDistinct"))`
D. `storesDF.select("division").dropDuplicates().count()`
E. `storesDF.select("division").distinct().count()`

Question 32

The code block shown below should return a new DataFrame with the mean of column **`sqft`** from DataFrame **`storesDF`** in column **`sqftMean`**. Choose the response that correctly fills in the numbered blanks within the code block to complete this task.

Code block:

```
storesDF.__1__(__2__(__3__).alias("sqftMean"))
```

A.
1. `agg`
2. `mean`
3. `col("sqft")`

B.
1. `mean`
2. `col`
3. `"sqft"`

C.
1. `withColumn`
2. `mean`
3. `col("sqft")`

D.
1. `agg`
2. `mean`
3. `"sqft"`

E.
1. `agg`
2. `average`
3. `col("sqft")`

Question 33

Which of the following code blocks returns the number of rows in DataFrame **storesDF**?

A. `storesDF.withColumn("numberOfRows", count())`
B. `storesDF.withColumn(count().alias("numberOfRows"))`
C. `storesDF.countDistinct()`
D. `storesDF.count()`
E. `storesDF.agg(count())`

Question 34

Which of the following code blocks returns the sum of the values in column **sqft** in DataFrame **storesDF** grouped by distinct value in column **division**?

A. `storesDF.groupBy.agg(sum(col("sqft")))`
B. `storesDF.groupBy("division").agg(sum())`
C. `storesDF.agg(groupBy("division").sum(col("sqft")))`
D. `storesDF.groupby.agg(sum(col("sqft")))`
E. `storesDF.groupBy("division").agg(sum(col("sqft")))`

Question 35

Which of the following code blocks returns a DataFrame containing summary statistics only for column **sqft** in DataFrame **storesDF**?

A. `storesDF.summary("mean")`
B. `storesDF.describe("sqft")`
C. `storesDF.summary(col("sqft"))`
D. `storesDF.describeColumn("sqft")`
E. `storesDF.summary()`

Question 36

Which of the following operations can be used to sort the rows of a DataFrame?

A. `sort()` and `orderBy()`
B. `orderby()`
C. `sort()` and `orderby()`
D. `orderBy()`
E. `sort()`

Question 37

The code block shown below contains an error. The code block is intended to return a 15 percent sample of rows from DataFrame **`storesDF`** without replacement. Identify the error.

Code block:

storesDF.sample(True, fraction = 0.15)

A. There is no argument specified to the **`seed`** parameter.

B. There is no argument specified to the **`withReplacement`** parameter.
C. The **`sample()`** operation does not sample without replacement — **`sampleby()`** should be used instead.
D. The **`sample()`** operation is not reproducible.
E. The first argument **`True`** sets the sampling to be with replacement.

Question 38

Which of the following operations can be used to return the top **`n`** rows from a DataFrame?

A. `DataFrame.n()`
B. `DataFrame.take(n)`
C. `DataFrame.head`
D. `DataFrame.show(n)`
E. `DataFrame.collect(n)`

Question 39

The code block shown below should extract the value for column **`sqft`** from the first row of DataFrame **`storesDF`**. Choose the response that correctly fills in the numbered blanks within the code block to complete this task.

Code block:

```
__1__.__2__.__3__
```

A.
1. `storesDF`
2. `first`
3. `col("sqft")`

B.
1. `storesDF`
2. `first`
3. `sqft`

C.
1. `storesDF`
2. `first`
3. `["sqft"]`

D.
1. `storesDF`
2. `first()`
3. `sqft`

E.
1. `storesDF`
2. `first()`
3. `col("sqft")`

Question 40

Which of the following lines of code prints the schema of a DataFrame?

A. `print(storesDF)`
B. `storesDF.schema`
C. `print(storesDF.schema())`
D. `DataFrame.printSchema()`
E. `DataFrame.schema()`

Question 41

In what order should the below lines of code be run in order to create and register a SQL UDF named **"ASSESS_PERFORMANCE"** using the Python function **assessPerformance** and apply it to column **customerSatistfaction** in table **stores**?

Lines of code:

1. `spark.udf.register("ASSESS_PERFORMANCE", assessPerformance)`
2. `spark.sql("SELECT customerSatisfaction, assessPerformance(customerSatisfaction) AS result FROM stores")`
3. `spark.udf.register(assessPerformance, "ASSESS_PERFORMANCE")`
4. `spark.sql("SELECT customerSatisfaction, ASSESS_PERFORMANCE(customerSatisfaction) AS result FROM stores")`

A. 3, 4
B. 1, 4
C. 3, 2
D. 2
E. 1, 2

Question 42

In what order should the below lines of code be run in order to create a Python UDF **`assessPerformanceUDF()`** using the integer-returning Python function **`assessPerformance`** and apply it to column **`customerSatisfaction`** in DataFrame **`storesDF`**?

Lines of code:

1. `assessPerformanceUDF = udf(assessPerformance, IntegerType)`
2. `assessPerformanceUDF = spark.register.udf("ASSESS_PERFORMANCE", assessPerformance)`
3. `assessPerformanceUDF = udf(assessPerformance, IntegerType())`
4. `storesDF.withColumn("result", assessPerformanceUDF(col("customerSatisfaction")))`
5. `storesDF.withColumn("result", assessPerformance(col("customerSatisfaction")))`
6. `storesDF.withColumn("result", ASSESS_PERFORMANCE(col("customerSatisfaction")))`

A. 3, 4
B. 2, 6
C. 3, 5
D. 1, 4
E. 2, 5

Question 43

Which of the following operations can execute a SQL query on a table?

A. `spark.query()`
B. `DataFrame.sql()`
C. `spark.sql()`
D. `DataFrame.createOrReplaceTempView()`
E. `DataFrame.createTempView()`

Question 44

Which of the following code blocks creates a single-column DataFrame from Python list years which is made up of integers?

A. `spark.createDataFrame([years], IntegerType())`
B. `spark.createDataFrame(years, IntegerType())`
C. `spark.DataFrame(years, IntegerType())`

D. `spark.createDataFrame(years)`
E. `spark.createDataFrame(years, IntegerType)`

Question 45

Which of the following operations can be used to cache a DataFrame only in Spark's memory assuming the default arguments can be updated?

A. `DataFrame.clearCache()`
B. `DataFrame.storageLevel`
C. `StorageLevel`
D. `DataFrame.persist()`
E. `DataFrame.cache()`

babin

Question 46

The code block shown below contains an error. The code block is intended to return a new 4-partition DataFrame from the 8-partition DataFrame **storesDF** without inducing a shuffle. Identify the error.

Code block:

```
storesDF.repartition(4)
```

babin

A. The **repartition** operation will only work if the DataFrame has been cached to memory.
B. The **repartition** operation requires a column on which to partition rather than a number of partitions.
C. The number of resulting partitions, 4, is not achievable for an 8-partition DataFrame.
D. The **repartition** operation induced a full shuffle. The **coalesce** operation should be used instead.
E. The **repartition** operation cannot guarantee the number of result partitions.

Question 47

Which of the following code blocks will always return a new 12-partition DataFrame from the 8-partition DataFrame **storesDF**?

A. `storesDF.coalesce(12)`
B. `storesDF.repartition()`
C. `storesDF.repartition(12)`
D. `storesDF.coalesce()`
E. `storesDF.coalesce(12, "storeId")`

Question 48

Which of the following Spark config properties represents the number of partitions used in wide transformations like **join()**?

A. `spark.sql.shuffle.partitions`
B. `spark.shuffle.partitions`
C. `spark.shuffle.io.maxRetries`
D. `spark.shuffle.file.buffer`
E. `spark.default.parallelism`

Question 49

In what order should the below lines of code be run in order to return a DataFrame containing a column **`openDateString`**, a string representation of Java's SimpleDateFormat?

Note that column **`openDate`** is of type **`integer`** and represents a date in the UNIX epoch format — the number of seconds since midnight on January 1st, 1970.

An example of Java's SimpleDateFormat is **`"Sunday, Dec 4, 2008 1:05 PM"`**.

A sample of **`storesDF`** is displayed below:

storeId	openDate
0	1100746394
1	1474410343
2	1116610009
3	1180035265
4	1408024997
...	...

Lines of code:

1. `storesDF.withColumn("openDateString", from_unixtime(col("openDate"), simpleDateFormat))`
2. `simpleDateFormat = "EEEE, MMM d, yyyy h:mm a"`
3. `storesDF.withColumn("openDateString", from_unixtime(col("openDate"), SimpleDateFormat()))`
4. `storesDF.withColumn("openDateString", date_format(col("openDate"), simpleDateFormat))`
5. `storesDF.withColumn("openDateString", date_format(col("openDate"), SimpleDateFormat()))`
6. `simpleDateFormat = "wd, MMM d, yyyy h:mm a"`

A. 2, 3
B. 2, 1
C. 6, 5
D. 2, 4
E. 6, 1

Question 50

Which of the following code blocks returns a DataFrame containing a column **`month`**, an integer representation of the month from column **`openDate`** from DataFrame **`storesDF`**?

Note that column **`openDate`** is of type integer and represents a date in the UNIX epoch format — the number of seconds since midnight on January 1st, 1970.

A sample of **`storesDF`** is displayed below:

storeId	openDate
0	1100746394
1	1474410343
2	1116610009
3	1180035265
4	1408024997
...	...

A. `storesDF.withColumn("month", getMonth(col("openDate")))`

B. `storesDF.withColumn("openTimestamp", col("openDate").cast("Timestamp")).withColumn("month", month(col("openTimestamp")))`

C. `storesDF.withColumn("openDateFormat", col("openDate").cast("Date")).withColumn("month", month(col("openDateFormat")))`

D. `storesDF.withColumn("month", substr(col("openDate"), 4, 2))`

E. `storesDF.withColumn("month", month(col("openDate")))`

Question 51

Which of the following operations performs an inner join on two DataFrames?

A. `DataFrame.innerJoin()`

B. `DataFrame.join()`

C. Standalone `join()` function

D. `DataFrame.merge()`

E. `DataFrame.crossJoin()`

Question 52

Which of the following code blocks returns a new DataFrame that is the result of an outer join between DataFrame **storesDF** and DataFrame **employeesDF** on column **storeId**?

A. `storesDF.join(employeesDF, "storeId", "outer")`
B. `storesDF.join(employeesDF, "storeId")`
C. `storesDF.join(employeesDF, "outer", col("storeId"))`
D. `storesDF.join(employeesDF, "outer", storesDF.storeId == employeesDF.storeId)`
E. `storesDF.merge(employeesDF, "outer", col("storeId"))`

Question 53

The below code block contains an error. The code block is intended to return a new DataFrame that is the result of an inner join between DataFrame **`storesDF`** and DataFrame **`employeesDF`** on column **`storeId`** and column **`employeeId`** which are in both DataFrames. Identify the error.

Code block:

```
storesDF.join(employeesDF, [col("storeId"), col("employeeId")])
```

A. The **`join()`** operation is a standalone function rather than a method of DataFrame — the **`join()`** operation should be called where its first two arguments are **`storesDF`** and **`employeesDF`**.
B. There must be a third argument to **`join()`** because the default to the how parameter is not **`"inner"`**.
C. The **`col("storeId")`** and **`col("employeeId")`** arguments should not be separate elements of a list — they should be tested to see if they're equal to one another like **`col("storeId") == col("employeeId")`**.
D. There is no **`DataFrame.join()`** operation — **`DataFrame.merge()`** should be used instead.

E. The references to **`"storeId"`** and **`"employeeId"`** should not be inside the **`col()`** function — removing the **`col()`** function should result in a successful join.

Question 54

Which of the following Spark properties is used to configure the broadcasting of a DataFrame without the use of the **`broadcast()`** operation?

A. `spark.sql.autoBroadcastJoinThreshold`
B. `spark.sql.broadcastTimeout`
C. `spark.broadcast.blockSize`
D. `spark.broadcast.compress`
E. `spark.executor.memoryOverhead`

Question 55

The code block shown below should return a new DataFrame that is the result of a cross join between DataFrame **`storesDF`** and DataFrame **`employeesDF`**. Choose the response that correctly fills in the numbered blanks within the code block to complete this task.

Code block:

```
__1__.__2__(__3__)
```

A.
1. `storesDF`
2. `crossJoin`
3. `employeesDF, "storeId"`

B.
1. `storesDF`
2. `join`
3. `employeesDF, "cross"`

C.
1. `storesDF`
2. `crossJoin`
3. `employeesDF, "storeId"`

D.
1. `storesDF`
2. `join`
3. `employeesDF, "storeId", "cross"`

E.
1. `storesDF`
2. `crossJoin`
3. `employeesDF`

Question 56

Which of the following operations performs a position-wise union on two DataFrames?

A. The standalone **`concat()`** function
B. The standalone **`unionAll()`** function
C. The standalone **`union()`** function
D. **`DataFrame.unionByName()`**
E. **`DataFrame.union()`**

Question 57

Which of the following code blocks writes DataFrame **`storesDF`** to file path **`filePath`** as parquet?

A. `storesDF.write.option("parquet").path(filePath)`
B. `storesDF.write.path(filePath)`
C. `storesDF.write().parquet(filePath)`
D. `storesDF.write(filePath)`
E. `storesDF.write.parquet(filePath)`

Question 58

The code block shown below contains an error. The code block is intended to write DataFrame **storesDF** to file path **filePath** as parquet and partition by values in column **division**. Identify the error.

Code block:

```
storesDF.write.repartition("division").parquet(filePath)
```

A. The argument **division** to operation **repartition()** should be wrapped in the **col()** function to return a Column object.
B. There is no **parquet()** operation for DataFrameWriter — the **save()** operation should be used instead.
C. There is no **repartition()** operation for DataFrameWriter — the **partitionBy()** operation should be used instead.
D. **DataFrame.write** is an operation — it should be followed by parentheses to return a DataFrameWriter.
E. The **mode()** operation must be called to specify that this write should not overwrite existing files.

Question 59

Which of the following code blocks reads a parquet at the file path **filePath** into a DataFrame?

A. `spark.read().parquet(filePath)`
B. `spark.read().path(filePath, source = "parquet")`
C. `spark.read.path(filePath, source = "parquet")`
D. `spark.read.parquet(filePath)`
E. `spark.read().path(filePath)`

Question 60

Which of the following code blocks reads JSON at the file path **filePath** into a DataFrame with the specified schema **schema**?

A. `spark.read().schema(schema).format(json).load(filePath)`
B. `spark.read().schema(schema).format("json").load(filePath)`
C. `spark.read.schema("schema").format("json").load(filePath)`
D. `spark.read.schema("schema").format("json").load(filePath)`
E. `spark.read.schema(schema).format("json").load(filePath)`